GSM Codec 4

4.1 OVERVIEW

This chapter describes the implementation of the Pan-European Digital Mobile Radio (DMR) Speech Codec Specification 06.10. This code was developed in accordance with the recommendation of the Conference of European Post and Telecommunications' (CEPT) Group Special Mobile (GSM). A copy of the recommendation can be obtained directly from this organization.

The recommendation describes how the software must perform, and provides a brief tutorial on the algorithm's operation. This chapter and the accompanying code were written to follow the structure of the recommendation.

For your reference, this chapter also includes subroutines for Voice Activity Detection (VAD, Specification 06.32) and Comfort Noise Insertion (CNI, Specification 06.12) . Together, these subroutines provide a more complete solution for GSM applications. For more information about these particular subjects, refer to the corresponding specifications.

4.1.1 Speech Codec

The speech codec for pan-European digital mobile radio is a modified version of a Linear Predictive Coder (LPC). The LPC algorithm uses a simplified model of the human vocal tract, which consists of a series of cylinders that vary in diameter. To produce voiced speech, you force air through these cylinders. You can represent this structure mathematically by a series of simultaneous equations that describe the cylinders.

Early LPC systems worked well enough for users to understand the coded speech, but often, not well enough to identify the speaker. The LPC system described in this chapter uses two techniques, Regular Pulse Excitation (RPE) and Long Term Prediction (LTP), to improve the quality of the coded speech. The improved speech quality is almost comparable to the speech quality produced by logarithmic Pulse Code Modulation (PCM).

The input to the speech codec is a series of 13-bit speech data samples sampled at 8 kSa/s. The codec operates on a 20 ms window (160 samples) and reduces it to 76 coefficients (260 bits) that result in a coded data rate of 13 kbits/s.

4.1.2 Software Comments

This section includes several comments that apply to the program examples in this chapter.

4.1.2.1 Multiply With Rounding

The GSM recommendation requires a *multiply with rounding* operation that provides biased rounding. Although the ADSP-21xx family does have a multiply with rounding instruction, this implementation does not use it because the instruction performs unbiased rounding (see the *ADSP-2100 Family User's Manual*), and the RND mode of the multiplier introduced bit-errors during the codec testing.

To eliminate this problem, the code uses a pre-multiply that stores the value H#8000 in the MR register. Unbiased rounding is then completed by a multiply/accumulate that produces the desired result. The MF register is loaded with H#80, and, at various points, an X-register is also loaded with H#80. Multiplying these two registers places the H#0000008000 in MR.

4.1.2.2 Arithmetic Saturation Results

The GSM recommendation also requires that arithmetic results be saturated. The ALU's AR_SAT mode easily accomplishes this task. Whenever an ALU operation produces an overflow, the output is automatically saturated at the appropriate value.

An arithmetic overflow occurs when the arithmetic operation produces an output that does not fit completely in the proper word size. In other words, the MSB of the word is not the sign bit. Since only the Most Significant Word (MSW) of a multiprecision value contains a sign bit, it is appropriate to check for overflow only in the MSW. When an LSW result does not fit in the output word size, it produces a carry into the next word, not an overflow.

When the LSW of a double precision result is produced, the saturation mode must be disabled. When the MSW is produced, the entire word can be checked for overflow, and saturated as necessary. Throughout the code, the ALU saturation mode is turned on when producing MSWs, or single precision values, and turned off for LSWs.

4.1.2.3 Temporary Arrays

The GSM recommendation specifies the creation of temporary arrays during codec execution. You do not need to save the value of these arrays, and whenever possible, they are eliminated in this implementation to save memory space. For example, the code overwrites the input speech window array with the output of the short term filter (difference signal d() array) instead of creating a new array.

In many cases, the code uses a single array for several purposes. The code's in-line comments indicate what information is stored by a particular section of code.

4.1.2.4 Shared Subroutines

The encoder is designed to produce an estimated signal based on the same information that is available at the decoder. This structure allows both systems to operate in synchronization. The encoder uses only the decoded values of transmitted parameters, insuring that it acts on the same information available to the decoder.

This requires that the encoder uses many of the same subroutines used by the decoder. Routines that are used by both systems are placed at the end of the listing, and are described only in the encoder section of this chapter.

4.2 ENCODER

Listing 4.1, *GSM0610.DSP*, is a full-duplex codec program example that contains the encoder and decoder subroutines. The encoder has three main sections:

- The linear prediction coder (LPC)–The LPC computes a set of eight reflection coefficients that describe the entire window of data.
- The regular pulse excitation (RPE) grid selector–The RPE grid selector breaks the input window into 4 sub-windows and computes a different excitation signal for each. By using 4 separate excitation signals, the codec can process speech signals that may change within a given window.
- The long term prediction (LTP) system–The LTP system reduces the error of the signal over the entire window.

4.2.1 Down Scaling & Offset Compensation Of The Input

The LPC encoder requires 160 samples of left-justified linear data as input. This window of data must be downshifted three bits, then upshifted two bits. The final result of this is to divide each value in half and set its two LSBs to zero. The first two instructions of the *offset_comp* loop perform this operation.

A double-precision high-pass filter is applied to the downshifted input to produce an offset-free signal. The code must execute a double-precision multiplication to maintain the necessary accuracy.

The rest of the *offset_comp* loop implements this filter. The shift instruction isolates the MSW of *L_z2*, which is held in the MR register. The AR register holds the LSW of *L_z2*. The LSW is multiplied by alpha (MY0) to produce the result *temp*. The new value of *L_s2* is generated, shifted into position and added to *temp*. After the addition of these two values, the MSW is multiplied by alpha and added to *L_s2* to produce *L_z2*.

The last steps of the loop compute the rounded value that is stored as output, and loads several registers for the next iteration. As in most of these operations, the compensation is performed in place, to conserve memory.

4.2.2 Pre-Emphasis Filtering

Before the LPC coefficients are determined, the input data is filtered by a first-order FIR filter. While filtering, the window is searched for the maximum value. This is necessary to ensure that the data can be properly scaled for the auto-correlation that follows. The *pre_emp* loop filters the input data.

This filter multiplies the delayed value and the filter coefficient, then adds the product to the current sample. The subroutine uses the SB register to check each sample for the number of redundant sign bits present. When the loop is completed, SB holds the negative number that corresponds to the number of growth bits in the maximum value of the window. The last step of the loop saves each output sample (written over the input), and prepares the MR register for the next multiply with round operation.

4.2.3 Auto-Correlation

The program uses the *auto_corr* loop for auto-correlation of the filtered input window to calculate the reflection coefficients for the entire window. To prevent an overflow during this procedure, the input data is scaled appropriately.

To compute the scale factor, the subroutine searches the input window for the maximum value, and determines the number of redundant sign bits (growth bits). The window is multiplied by a scale factor to insure that there are three redundant sign bits to handle any growth during the auto-correlation. The search operation is completed in the previous filtering section. The code loop labeled *scale* adjusts the data to ensure the necessary number of growth bits.

The *corr_loop* loop determines the first nine terms of the auto-correlation sequence. The auto-correlation is the sum of the products of the signal with itself offset for k = 0–8. The terms of the sequence are used to compute the reflection coefficients.

The auto-correlation code sets two pointers to the data areas (I1, I5), one pointer to the output array (I6), and uses another pointer as a down-counter for the inner loop (*data_loop*). Since the inner loop executes one less time for each successive value of the auto-correlation sequence, the CNTR is set to I2 for each new auto-correlation term.

After *data_loop* is completed, the next term of the sequence is in the MR register. This value is saved in the output array after incrementing the pointer to the data array, and decrementing the down-counter.

When *corr_loop* is completed, all nine terms of the auto-correlation sequence have been generated and stored in the double precision array L_ACF(). The input data is rescaled by the *rescale* loop before the reflection coefficients are computed.

4.2.4 The Schur Recursion

The theory behind any LPC voice coder is that the throat can be modeled as a series of concentric cylinders with varying diameters. An excitation signal is passed through these cylinders, and produces an output signal. In the human body, the excitation signal is air moving over the vocal cords. In a digital system, the excitation signal is a series of pulses input to a lattice filter with coefficients that represent the sizes of the cylinders.

An LPC system is characterized by the number of cylinders it uses for the model. The DMR system uses eight cylinders, therefore, eight reflection coefficients must be generated. This system uses the *Schur recursion* to efficiently solve for each coefficient.

After a coefficient is determined, two equations are re-computed and used to solve for the next coefficient. The following equations are used:

$$r(n) = \frac{\mathrm{ABS}[P(1)]}{P(0) \times \mathrm{SIGN}[P(1)]} \qquad \text{for } n = 1-8$$

$$P(0) = P(0) + P(1) \times r(n)$$

$$P(m) = P(m+1) + r(n) \times K(9-m) \qquad \text{for } m = 1-8-n$$

$$K(9-m) = K(9-m) + r(n) \times P(m+1)$$

The P() and K() arrays are initialized with values from the auto-correlation sequence determined earlier. If during the computation, the value of ABS[P(1)] ÷ P(0) is greater than or equal to one, all r-values are set to zero, and the program proceeds with the transformation of the r-values to Logarithmic-Area-Ratios (LARs) described in the next section.

Before initializing the P() and K() arrays, the double precision auto-correlation sequence L_ACF() is normalized. The *set_acf* loop normalizes each of the nine values and places them in the array acf(). The SE register is initialized before entering the loop by the EXP instruction of the shifter. The first value of the auto-correlation sequence is always the largest value of the sequence. The normalization of the rest of the sequence is based on the number of redundant sign bits in the first value.

The *create_k* loop copies the values of the normalized auto-correlation sequence acf() into the appropriate locations in the P() and K() arrays.

The *compute_reflec* loop actually implements the Schur recursion. The I2 and I3 pointers are set to the beginning of the two arrays used to compute the r-values. The absolute values of P(1) and P(0) are compared. If the divide produces an invalid result (r > 1), the code executes a JUMP instruction to skip the remaining computations. Since this test is also performed after the exit from this loop (and since the P() array is not altered if the JUMP is executed) the program eventually jumps to the *zero_reflec* code block, and sets each r-value to zero.

If the divide is valid, it is computed with the ADSP-2100 family divide instructions. The DIVS command computes the sign bit of the quotient, and 15 DIVQs compute the remaining bits. These commands produce the 16-bit value in the AY0 register. After the division, another test is performed to see if the original dividend and divisor are equal (the division instruction does not saturate), if so, the quotient is saturated to 32767. The sign of the quotient is determined from the original sign of P(1), and the r-value is stored in the result array.

The new value for P(0) is computed according to the equation shown above. The two equations are re-computed in the *schur_recur* loop. The counter for this loop is set from the I6 register, which is used as a down-counter.

The *compute_reflec* loop generates the first seven reflection coefficients. The eighth r-value is computed outside of the loop. The code outside the loop is identical to the code inside, but it is not included in the loop since the K() and P() arrays do not need to be re-calculated after the final r-value is computed.

4.2.5 Transformation Of The Reflection Coefficients

The reflection coefficients generated by the Schur recursion are constrained to be in the range -1 < r() < 1. To produce a value that can be more easily quantized into a small number of bits, the following equation transforms the reflection coefficients to Logarithmic-Area-Ratios (LARs): This transformation process is similar to logarithmic companding used in

$$\mathrm{LAR}(i) = \mathrm{Log}_{10} \frac{1 + r(i)}{1 - r(i)}$$

log-PCM coding. Taking the logarithm of a number in a fixed precision n-bit machine allocates more bits for the smaller values, and tends to saturate for larger values.

In the implementation of the encoder, the logarithm is approximated with a linear segmentation (as in log-PCM) to simplify the computation. Instead of the divide and logarithm operations, the segmentation simplifies to multiplies, adds, and compares.

The code that transforms the reflection coefficients starts at label *real_rs*. The *compute_lar* loop executes once for each r-value, and produces one LAR-value for each iteration. The three values that *temp* can become are computed first, and stored in various registers. The final ELSE value is left in AR, which holds the result. The inner IF statement is checked, and if true, AR is set with the appropriate *temp* value.

The first IF statement is checked last. This ensures that AR holds the correct value for *temp*. The last step of the loop generates the sign value for *temp*, and stores the LAR value.

4.2.6 Quantization & Coding Of The Logarithmic-Area-Ratios

The LARs produced in the last section of the program must be quantized and coded into a limited number of bits for transmission. The *quantize_lar* loop computes the following equation to generate the coded LARs or LAR_Cs.

$$LAR_c(i) = N\,int\left[A(i) \times LAR(i) + B(i)\right]$$

The function Nint defines the nearest integer value to its input. Since each LAR has a different dynamic range, they are coded into varying word sizes. Using a table, the values for A() and B() are defined to reflect these differences. In addition to A() and B(), the table defines the maximum and minimum values for each LAR_C. After each $LAR_C()$ is computed, it is saturated at the appropriate value.

To implement this coding in the program, several Index (I) registers are set to data arrays representing a table. The AX0 register is set to 256 and is used for rounding the results within the loop.

The code is a straightforward implementation of the recommendation. The first multiply computes $A() \times LAR()$, and the value for B() is added to the product. This sum, which is rounded by the addition of AX0, is downshifted nine bits for saturation. After limiting, the minimum value is subtracted from the final value to produce the $LAR_C()$ that is transmitted.

The eight LAR_Cs are copied from their array to the *xmit_buffer* that holds the entire window of 76 coefficients to be transmitted. A similar transfer is executed every time some of the code words are available for transmission.

4.2.7 Decoding Of The Logarithmic-Area-Ratios

The LARs that were just coded are now decoded (using the *decode_larc subroutine*), and used in the short term analysis section. The encoder uses the decoded LARs because that information matches the information that the receiving decoder uses. This lets the encoder and decoder produce results based on the same data.

The decoded LARs (or LAR_{pp}) are calculated from the coded LARs (LAR_cs) with the following equation:

$$LAR_{pp}(i) = \frac{LAR_c(i) - B(i)}{A(i)}$$

To simplify the implementation of this equation, a table in memory contains the reciprocal of A(i). The equation becomes a subtraction and a multiply, which is faster than a divide.

The same decoding subroutine is used in the encoder and decoder, so the code is written as a separate subroutine that can be called from either routine. The *decode_larc* subroutine is located near the end of the listing.

This subroutine is a straightforward implementation of the recommendation. The minimum value for the current LAR_c (from the table) is added to the coded LAR_c. This value is upshifted ten bits, and B() (upshifted one bit) is subtracted. This remainder is multiplied by the reciprocal of A(). The final value is doubled before being stored in the LAR_{pp}() array.

4.2.8 Short Term Analysis Filtering

Once the LARs are decoded, they are transformed back into reflection coefficients and used in an 8-pole lattice filter. The short term analysis filter uses the input speech window and reflection coefficients as inputs, and produces a difference signal as output. The difference signal represents the difference between the actual input speech window, and the speech that would be generated based only on the reflection coefficients.

The difference signal is used by the long term predictor (LTP) section of the codec. The LTP is described in Section 4.2.9.

To avoid transients that could occur with a rapid change of filter coefficients, the LARs are linearly interpolated with the previous set of LARs. The input speech frame is broken into four sections (not at the same boundaries as sub-windows), and a different set of interpolated coefficients is used for each section. A table defines the coefficients that are used for each section of the speech frame.

When the interpolated LAR value is generated for each section, it must be transformed from a Logarithmic-Area-Ratio back into a reflection coefficient. This sequence must also be performed in the decoder. To minimize code, the *st_filter* subroutine, called by the encoder and decoder, interpolates, transforms, and executes the short term filter for each section of the input frame.

This subroutine is similar for the encoder and decoder except that different 8-pole lattice filters are called for the encoder and decoder. This is easily coded as an indirect call through one of the index registers. Register I6 is set to the address of *st_analysis* (for the encoder) and the indirect *call (I6)* instruction jumps to that subroutine.

The LARs are interpolated at four points in the *st_filter* routine. The first section's coefficients are interpolated by the *k_end_12* loop. Every *k_end_xx* code loop uses the *old_larpp* array (pointed to by I4) and the *larpp* array (the current decoded LARs) to produce a weighted sum of the two, and stores the output in the array *larp*. The *larp* array is transformed into reflection coefficients that are used by the short term filter.

4.2.8.1 Transformation Of The LARs Into Reflection Coefficients

Before transmission, the computed reflection coefficients are transformed into LARs to provide favorable quantization characteristics. Although this transformation is useful for transmission, the LARs must be transformed back into reflection coefficients before they can be used as inputs to the synthesis filter.

The *make_rp* subroutine transforms the LARs back into reflection coefficients and stores them in the rp() array. This subroutine's implementation is similar to the subroutine that codes the LARs. The result for each IF-THEN-ELSE test is created first, with the final ELSE value stored in the AR register. The condition of each IF statement is tested from the inside out. The final test of the loop generates the sign of the output. The rp() array is stored in program memory for easy fetching during the filtering subroutine.

4.2.8.2 Short Term Analysis Filtering

The short term analysis filter implements a lattice structure by solving the following five equations:

1) $d_0(k) = s(k)$

2) $u_0(k) = s(k)$

3) $d_i(k) = d_{i-1}(k) - r'_i \times u_{i-1}(k-1) \quad \text{with } i = 1-8$

4) $u_i(k) = u_{i-1}(k-1) + r'_i \times d_{i-1}(k) \quad \text{with } i = 1-8$

5) $d(k) = d_8(k)$

The *st_analysis* subroutine computes the five equations shown above. Several registers are setup before calling this subroutine. The CNTR register is set with the number of output samples to be generated during this call. The *st_compute* loop executes once for each output sample created generated. Pointers to the rp() coefficient and u() delay line are setup, and the input sample is fetched.

The *st_loop* loop calculates the two iterative equations (3 and 4) shown above. The first multiply prepares the MR register and loads the coefficient and delay values. The second and third lines of the loop generate a new u_i() value (equation 4). The fourth line saves the previous value of u() (for use in the next iteration) and prepares the MR register. The final two lines generate a new d_i() value (equation 3) that is held in the AR register.

When the *st_loop* is exited, the value for d_8(k) is in the AR register. This value is stored in the output array, and the loop re-executes as necessary.

4.2.9 Calculation Of The Long Term Parameters

The long term calculations of the LPC speech codec are performed four times for each window of data. The calculations are the same for each sub-window, so they are implemented as a set of subroutines that are called four times per frame.

Once the calculations are complete for a sub-window, the 17 coefficients (Nc, bc, mc, xmaxc, and xMc[0–12]), which are stored contiguously, are copied to the *xmit_buffer*. Since the previous sub-window's coefficients do not need to be saved, the same memory locations are used by the next sub-window.

The code must set the I3 register to the input array before the first call to the subroutines. The I3 register is automatically incremented by the necessary number (40) during the *lt_analysis* section of code.

4.2.9.1 Long Term Analysis Filtering

The long term predictor (LTP) produces two coefficients to describe each sub-window. A long term correlation lag (Nc) represents the maximum cross-correlation between samples of the current sub-window and the previous two sub-windows. A gain parameter (bc) represents the quantized ratio of the power of delayed samples to the maximum cross-correlation value.

The value for Nc is determined by computing the cross-correlation between the short-term residual signal of the current sub-window and the signal of the previous sub-windows. The *cross_loop* loop computes each value of the cross-correlation and puts the maximum lag in AX1.

The transmitted value of Nc is not coded, but sent using a 7-bit word.

The coded value for bc is determined using the *table_dlb* lookup-table. This table holds values that indicate the ratio of the numbers. The coded value of bc is the index into a table that satisfies the relationship.

The *ltp_computation* subroutine searches the input sub-window for a maximum value. When the *find_dmax* code loop is exited, SB holds a negative number that corresponds to the number of redundant sign bits present in the maximum value of the sub-window.

The *init_wt* loop uses the value determined above, and shifts the data to ensure that there is at least six redundant sign bits for growth during the cross-correlation execution.

The execution of the cross-correlation is similar to the execution of the auto-correlation performed for the Schur recursion. The only difference is that the auto-correlation uses the same signal for both inputs, while the cross-correlation uses two different signals, dp() and wt(). Each term of the cross-correlation is checked, and if it exceeds the current maximum, the new value is taken as the maximum, and its index is saved as Nc. When the *cross_loop* loop is exited, the value in AX1 is the final value of Nc.

The *power* loop determines the power of the maximum cross-correlation and the gain (bc) value. The value for bc is the ratio of the power of the cross-correlation and the maximum value of the correlation. This ratio is expressed as one of the four values in *table_dlb*, which is stored in data memory. The transmitted value for bc is the index into the table that satisfies the relationship.

4.2.9.2 Long Term Synthesis Filtering

The short-term analysis filter computes a residual signal and stores it in the d() array. Using the LTP coefficients determined by this filter, an estimated short-term residual signal, stored in the dpp() array, is computed from the previously reconstructed short-term residual samples from the dp() array and the new Nc and bc parameters.

From the values of the dpp() array, the long-term residual signal is computed and stored in the e() array. The e() array will be applied to a FIR filter to generate the residual pulse excitation (RPE) signal.

4.2.10 Residual Pulse Excitation Encoding Section

After the long-term residual signal is produced, it is sent through a FIR filter to generate an excitation signal for the sub-window. After decimation, the maximum excitation sequence is determined and coded for transmission.

An Adaptive Pulse Code Modulation (APCM) technique codes the sequence. The maximum value in the sequence is determined and logarithmically coded into six bits. The sequence is normalized and uniformly coded into three bits.

4.2.10.1 Weighting Filter

The output of the long term analysis filtering section, e(), is applied as an input to an FIR filter. The filter's coefficients are stored in a table. This section of code uses a special "block" filter that produces the 40 central samples of a conventional filter. The x() output array is used in the RPE grid selector described in the following section.

The *compute_x_array* loop implements the FIR block filter. The e() input array is placed into the wt() temporary array with five zeros padded at each end. The zero padding is necessary because the block filter implementation tries to use values outside of the defined range of e().

Pointers to the input and output arrays are initialized and the code enters the *compute_x_array* loop. The first two operands of the convolution are fetched, and the appropriate rounding value is placed in the MR register. An inner loop is executed to compute the convoluted output value.

The final double precision output value must be scaled by four before the MSW is stored. This is accomplished using two double-precision additions. After the first addition, the AV (overflow) flag is checked. If an overflow occurs, the output value is saturated and the second addition is skipped. The MS part of the second addition is performed with the saturation mode of the ALU enabled, which automatically causes saturation if an overflow occurs.

4.2.10.2 Adaptive Sample Rate Decimation By RPE Grid Selection

The output of the weighting filter, put in the x() array, is examined to determine the excitation sequence that is used. The x() array is decimated into four sub-sequences. The sub-sequence with the maximum energy is used as the excitation signal, and the value of m indicates the RPE grid selection. The following formula performs the decimation:

$$x_m(i) = x(m + 3 \times 1)$$
$$\text{where } i \quad = 0 - 12, \quad m = 0 - 3$$

The *find_mc* loop determines the sub-sequence with the maximum energy. The energy of each X_m() array is determined by the *calculate_em* loop. This loop multiplies each element of the sequence (downshifted twice) by itself and computes the sum. The value of m that indicates the sub-sequence with the maximum energy is held in AX0.

Once the *find_mc* loop is completed, the value for mc is stored, and the appropriate sub-sequence is copied into the wt() array. The code then determines the maximum element of the x_m() array and holds it in the AR register for quantizing.

4.2.10.3 APCM Quantization Of The Selected RPE Sequence

The maximum value of the sequence is coded logarithmically using six bits. The upper three bits of *xmaxc* hold the exponent of *xmax*, and the lower three bits hold the mantissa. Once *xmax* is coded, the array can be normalized without performing a division.

The x_m() array is normalized by downshifting each element by the exponent of *xmaxc*, and multiplying it by the inverse of the *xmaxc*'s mantissa. The normalized array is uniformly quantized with three bits.

The *quantize_xmax* loop performs the logarithmic quantization of *xmax* by determining the exponent and mantissa, and then positioning them appropriately. The call to *get_xmaxc_pts* decodes *xmaxc*, then returns to the calling routine with the exponent and mantissa of *xmax*.

The *compute_xm* loop performs the normalization of x_m(). The inverse of *xmax*'s mantissa is read from a table and stored in MY0, while the magnitude of the downshift is stored in SE. After normalization, the upper three bits of the result are biased by four, and stored in the x_mc() array for transmission.

4.2.10.4 APCM Inverse Quantization & RPE Grid Positioning

The x_mc() array must be decoded for use as the excitation signal. The subroutine *rpe_decoding* is used by the encoder and decoder. This subroutine assumes that the coded mantissa of *xmaxc* is available in MX0, and its exponent is in AY1.

The actual value for the mantissa is read from *table_fac* and stored in MY0, while the adjusted exponent is stored in SE and the value of *temp3* is placed in AY1. Various pointers are initialized before entering the *inverse_apcm* loop, which decodes the entire x_mc() array. After decoding each element, it is stored in the x_mp() array.

The ep() array is reconstructed from the decoded x_mc() array. The ep() array is first set to zero over its entire length, then filled with the interpolated, decoded values of the x_mc() array. The intermediate x_mp() array is not used.

4.2.10.5 Update Of The Reconstructed Short Term Residual Signal

The final step of the encoder's sub-window computation is to update the short term residual signal, dp(). The process involves updating the array and computing the new short term residual signal based on the reconstructed long term residual signal and the long term analysis signal. Both of these steps are completed by the *update_dp_code* loop.

The *update_dp* loop updates the dp() array by delaying the data one sub-window. The *fill_dp* loop adds the dpp() array, generated by the long term analysis filter, and ep(), the reconstructed long term residual signal, then stores the result at the end of the dp() array.

4.3 DECODER

Many of the sections in the decoder are also contained in the encoder, so they have already been described. The three sections unique to the decoder are the long term synthesis filter, the short term synthesis filter, and the post processing. Variables that are unique to the decoder and must be stored between calls have an “r” in their names, such as drp().

The decoder for the LPC speech codec creates an excitation signal for the short term synthesis filter. The excitation window is created using the 17 sub-window coefficients that were generated by the encoder. The excitation signal is used as input to a lattice filter with coefficients of the eight decoded LAR_Cs. The output of this filter is a full window of speech data. The speech window is down-scaled and sent through a de-emphasis filter before returning.

The *dmr_decode* subroutine computes the output speech window from the 76 input coefficients. The *recv_data* subroutine copies coefficients from the input buffer to the appropriate location in memory. The transmitted LAR_Cs are copied into their array and decoded using the *decode_larc* routine described in section 4.2.8. These values are used by the short term synthesis filter described below.

Computation of the sub-window data starts by copying the sub-window coefficients into their arrays. A call to *get_xmaxc_pts* breaks the coded value of xmaxc into its two parts for use by the *rpe_decode* routine (see section 4.2.10.4). The *lt_predictor* routine takes the reconstructed ep() array and computes the new values for the short term reconstructed residual signal drp(). Four calls to these subroutines are executed to compute the excitation signal for the short term synthesis filter.

The *post_process* loop completes the computation of the output window, then control is returned to the calling routine.

4.3.1 Short Term Synthesis Filtering

The decoder uses short term synthesis filtering that is almost identical to the encoder’s short term synthesis filtering. The *st_filter* routine is called, but with different parameters. The I6 register is set to the address of *st_synthesis*, the lattice filter used by the decoder, and register I4 is set to the address of *old_larpp*, the array that holds the previous LARs for the decoder. Address register I0 points to a temporary array that holds the reconstructed short term residual signal that was generated for each sub-window.

Section 4.2.9.1 has a complete description of the *st_filter* routine. Section 4.2.9.2. describes the transformation of LARs into reflection coefficients.

4.3.1.1 Short Term Synthesis Filter

The short term synthesis filter is an implementation of an 8-pole lattice filter. It uses the reconstructed short term residual signal as an excitation, and computes the reconstructed speech signal as output. LARs that are averaged and transformed are used as the coefficients for the filter.

The lattice filter used in the decoder is different from the filter used in the encoder. It is defined by the following five equations.

1) $sr_0(k) = dr'(k)$

2) $sr_i(k) = sr_{i-1}(k) - rr'_{(9-i)} \times v_{8-i}(k-1) \quad \text{with } i = 1-8$

3) $v_{9-i}(k) = v_{8-i}(k-1) + rr'_{(9-i)} \times sr_i(k) \quad \text{with } i = 1-8$

4) $sr'(k) = sr_8(k)$

5) $v_0(k) = sr_8(k)$

The code that solves these equations is contained in the subroutine *st_synthesis*. The *st_synth_compute* loop generates one output value (sr) during each pass of the loop, while *st_synth_loop* recursively solves the two inner equations.

The first two instructions of the *st_synth_loop* loop generate a new value for $sr_{(i)}$. The next three instructions generate the new value for $v_{(9-i)}$. The address modification that points to the v() array uses a non-sequential modifier.

The first fetch to the v() array reads v(7) and points to v(6). The first fetch in the loop reads v(6) and modifies the pointer to v(8). The last instruction of the loop writes to the v() array, places the updated value in v(8), and modifies the pointer to v(5) for the next read. After the *st_synth_loop* is exited, the code must modify the pointer so the next write is to v(0).

4.3.2 Long Term Synthesis Filtering

The long term synthesis filtering used in the decoder takes the lag (Nc), gain (bc), and reconstructed long term residual signal in ep() and generates the reconstructed short term residual signal in drp(). This signal is used as an input to the short term filter.

The received lag coefficient is checked to ensure that a transmission error did not cause an inappropriate value to be received. If the value falls outside its permissible range, it is set to the previous value. The decoded gain value is multiplied by the previous reconstructed short term residual signal (drp()) and subtracted from the reconstructed long term residual signal (ep()) to generate the reconstructed short term residual signal for the current sub-window. Also, the drp() array is updated by the subroutine.

The *compute_drp* loop generates the new set of reconstructed short term residual values, and *update_drp* updates (or delays) the values of the drp() array.

4.3.3 Post Processing

The final stage of the decoder involves the de-emphasis filtering and down scaling. These two operations are performed by the *post_process* loop. A first order IIR filter is applied to the output of the short term synthesis filter. The first two instructions of the loop accomplish this while the next two instructions double the value of the output.

The last two instructions mask the three LSBs of the output, and store the final value in the output array.

4.4 BENCHMARKS & MEMORY REQUIREMENTS

The following listings implement the entire set of GSM 06 series speech functions on the ADSP-2101. This code is validated to pass all available GSM test vectors. This code is also available on the diskette included with this book.

Table 4.1 presents benchmarks for the system that include encoding and decoding, voice activity detection, comfort noise insertion and generation, and discontinuous transmission functions. The ADSP-2100 family instruction set lets you code the entire set of GSM speech functions into 1988 words of program memory and 964 words of data memory. All the code fits in the internal memory of the ADSP-2101 or the ADSP-2171 microcomputer.

These benchmarks are for ADSP-2101 (13 MHz instruction rate) and ADSP-2171 (26 MHz instruction rate) GSM systems with a 20 ms frame. Most of the time in the frame is unused, leaving ample time and processing power to implement additional features, such as acoustic echo cancellation.

	Cycle Count (maximum worst case)	*Time Required (ms)*	*Processor Loading (%)*
ADSP-2101 (13 MHz)			
RPE-LTP LPC Encoder	49300	3.8	19.0
RPE-LTP LPC Decoder	14400	1.1	05.5
Voice Activity Detector	02141	0.17	00.9
Total of 06 series functions	65841	5.07 ms	25.4 %
Free			74.6 %
ADSP-2171 (26 MHz)			
RPE-LTP LPC Encoder	49300	1.9	9.5
RPE-LTP LPC Decoder	14400	0.55	2.75
Voice Activity Detector	02141	0.09	0.45
Total of 06 series functions	65841	2.54 ms	12.7 %
Free			87.3 %

Table 4.1 GSM Implementation Benchmarks

4.5 LISTINGS

This section contains the listings for this chapter.

4 GSM Codec

```
{______________________________________________________________________________
GSM_RSET.DSP
                  Analog Devices Inc.  DSP Division
                  One Technology Way, Norwood, MA  02062
                  DSP Applications: (617) 461-3672

   This routine performs all of the necessary initialization of variables
   in all of the various GSM speech processing routines. All of these
   variables are defined in RAM, in either Program or Data Memory.

   The subroutine “reset_codec” must be called following DAG initialization
   after system power-up or system reset, before any other subroutine is
   called and before the data acquisition routine is enabled.

   This program must also be called to set the initial state prior to
   validation with the GSM test vectors.

   ADSP-2101 Execution cycles:      894 maximum

Release History:
__Date___   _Ver_ _______________Comments_________________
01-Sep-89   58    Initial implementation
10-Jan-90   1.00  Second Release
01-Nov-90   2.00  Third release
______________________________________________________________________________}

.MODULE     software_reset;
.ENTRY      reset_codec;

{  from 06.10 (encoder/decoder) and 06.12 (comfort noise in encoder)
         and 06.31 (dtx in encoder) }

.EXTERNAL   u, dp, nrp;
.EXTERNAL   oldlar_buffer, oldxmax_buffer, cni_wait;
.EXTERNAL   speech_count, oldlar_pntr, oldxmax_pntr;
.EXTERNAL   old_LARrpp, old_LARpp;
.EXTERNAL   drp, mp, L_z2_l, L_z2_h;
.EXTERNAL   z1, msr, v;

{  from 06.32 (voice activity detection) }

.EXTERNAL   rvad, normrvad, L_sacf, L_sav0;
.EXTERNAL   pt_sacf, pt_sav0, L_lastdm;
.EXTERNAL   oldlagcount, veryoldlagcount;
.EXTERNAL   e_thvad, m_thvad, adaptcount;
.EXTERNAL   burstcount, hangcount, oldlag;

{  from 06.31 (dtx codeword decoding) and 06.11 (sub and mute) }
```

```
.EXTERNAL   valid_sid_buffer, sub_n_mute, sid_inbuf, taf_count;

{  from 06.12 (comfort noise in decoder) }

.EXTERNAL   seed_lsw, seed_msw;

{  from shell }

.EXTERNAL   speech_1, speech_2, coeff_codeword;

reset_codec:AX0 = 0;

         I0  = ^L_sacf;
         CNTR = 54;
         CALL zero_dm;

         I0  = ^L_sav0;
         CNTR = 72;
         CALL zero_dm;

         I0  = ^speech_1;
         CNTR = 160;
         CALL zero_dm;

         I0  = ^speech_2;
         CNTR = 160;
         CALL zero_dm;

         I0  = ^drp;
         CNTR = 160;
         CALL zero_dm;

         I4  = ^dp;
         CNTR = 120;
         CALL zero_pm;

         I0  = ^msr;          { msr, old_LARrpp[8], v[9] }
         CNTR = 18;
         CALL zero_dm;

         I0  = ^u;            { u[8], oldLARpp[8], z1, L_z2_h, L_z2_l, mp }
         CNTR = 20;
         CALL zero_dm;

         I0  = ^L_lastdm;     { L_lastdm[2], oldlagcount, veryoldlagcount, }
         CNTR = 6;            { adaptcount, burstcount }
         CALL zero_dm;
```

(listing continues on next page)

```
I0  = ^sub_n_mute;   { sub_n_mute, sid_inbuf }
CNTR = 2;
CALL zero_dm;

DM(coeff_codeword) = AX0;

AX0 = 40;
DM(oldlag) = AX0;
DM(nrp) = AX0;

AX0 = 15381;
DM(seed_lsw) = AX0;
AX0 = 7349;
DM(seed_msw) = AX0;
AX0 = 1;
DM(speech_count) = AX0;
AX0 = -4;
DM(cni_wait) = AX0;

AX0 = -1;
DM(hangcount) = AX0;
AX0 = 20;
DM(e_thvad) = AX0;
AX0 = 31250;
DM(m_thvad) = AX0;
AX0 = -7;
DM(normrvad) = AX0;

AX0 = -24;
DM(taf_count) = AX0;

AX0 = ^L_sacf;
DM(pt_sacf) = AX0;
AX0 = ^L_sav0;
DM(pt_sav0) = AX0;
AX0 = ^oldlar_buffer;
DM(oldlar_pntr) = AX0;
AX0 = ^oldxmax_buffer;
DM(oldxmax_pntr) = AX0;

I0  = ^rvad;
AX0 = 24576;
DM(I0,M1) = AX0;
AX0 = -16384;
DM(I0,M1) = AX0;
AX0 = 4096;
DM(I0,M1) = AX0;
AX0 = 0;
CNTR = 6;
CALL zero_dm;
```

```
         I0  = ^valid_sid_buffer;
         AX0 = 42;
         DM(I0,M1) = AX0;
         AX0 = 39;
         DM(I0,M1) = AX0;
         AX0 = 21;
         DM(I0,M1) = AX0;
         AX0 = 10;
         DM(I0,M1) = AX0;
         AX0 = 9;
         DM(I0,M1) = AX0;
         AX0 = 4;
         DM(I0,M1) = AX0;
         AX0 = 3;
         DM(I0,M1) = AX0;
         AX0 = 2;
         DM(I0,M1) = AX0;
         AX0 = 0;
         DM(I0,M1) = AX0;

         RTS;

zero_dm: DO dmloop UNTIL CE;
dmloop:     DM(I0,M1) = AX0;
         RTS;

zero_pm: DO pmloop UNTIL CE;
pmloop:     PM(I4,M5) = AX0;
         RTS;

.ENDMOD;
```

Listing 4.1 Initialization Routine (GSM_RSET.DSP)

4 GSM Codec

```
{ GSM0610.DSP

These subroutines: dmr_encode and dmr_decode, represent a full duplex codec
for the Pan-European Digital Mobile Radio Network. The code implements a
Linear Predicitive Coder (LPC) which incorporates a Long Term Predictor
with Regular Pulse Excitation (LTP-RPE), as defined by the CEPT/GSM 06.10
specification. This code also includes support for the DTX functions of the
GSM specification. Calls are made to Voice Activity Detection (06.32) and
Comfort Noise Insertion (06.12) subroutines. This code has been verified
and successfully transcodes the GSM 06.10 Test Sequence Version 3.0.0 dated
April 15, 1988. The -Dnovad switch must be used at assembly to turn of
Voice Activity Detection during validation. In-line comments refer to
various sections of this recommendation. It is assumed that the reader is
familiar with that document.

Release History:
   03-Feb-89 32 Initial release.
   20-Jun-89 56 Fix reflect coef sect to pass all 3.0.0 vectors.
   10-Jan-90 1.00 Second release.

Information furnished by Analog Devices is believed to be accurate and
reliable. However, no responsibility is assumed by Analog Devices for its
use; nor for any infringement of patents or other rights of third parties
which may result from its use. Portions of the algorithms implemented in
this code may have been patented; it is up to the user to determine the
legality of their application.

Assembler Preprocessor Switches:
   -cp switch          must always be used when assembling
   -Dnovad switch      disables VAD for validation of 06.10
   -Dalias switch      aliases some variables to save RAM space
   -Ddemo switch       enables several functions necessary for
                       the eight-state demonstration

Calling Parameters:
   I0 —> Input Speech Buffer (for dmr_encode)
   I1 —> Coefficient Buffer (for both)
   I2 —> Output Speech Buffer (for dmr_decode)
   AX0 -> Silence Descriptor Frame flag (for dmr_decode)
   M0=0; M1=1;    M2=-1; M3=2;
   M4=0; M5=1;    M6=-1;
   L0=0; L1=0;    L2=0; L3=0;
   L4=0; L5=0;    L6=0; L7=0;

Return Values:
   I1 —> Coefficient Buffer (for dmr_encode)
   I2 —> Output Speech Buffer (for dmr_decode)
```

```
Altered Registers:
   AX0, AX1, AY0, AY1, AR, AF,
   MX0, MX1, MY0, MY1, MR, MF,
   SI, SE, SB, SR,
   I0, I1, I2, I3, I4, I5, I6
   M0, M7

ADSP-2101 Computation Time (without Voice Activity Detection):
Encoder  49300 cycles maximum
Decoder  14400 cycles maximum

State:                         Encoder     Decoder
   speech only                 46900       14000  cycles maximum
   comfort noise generation    47200       14400  cycles maximum
   speech hangover             49300       14000  cycles maximum
}

.MODULE/RAM/BOOT=0    Digital_Mobile_Radio_Codec;
.ENTRY      dmr_encode, dmr_decode, schur_routine, divide_routine;
.EXTERNAL   comfort_noise_generator;
.EXTERNAL   vad_routine, update_periodicity;
.EXTERNAL   vad, lags;

{___________________________Conditional Assembly______________________________}
{  Use (asm21 -cp -Dalias) to alias some variables to save RAM                }

#ifdef alias
.INCLUDE    <var0610.ram>;
   #define r dpp
   #define k dpp+25
   #define acf dpp+8
   #define p dpp+17
   #define LAR dpp+25
   #define rp wt
   #define LARp wt+8
   #define LARpp DPP
   #define LARc wt
   #define ep wt
   #define mean_larc dpp+17
#else
   .INCLUDE        <var0610.h>;
#endif
{_____________________________________________________________________________}

.INCLUDE <init0610.h>;

{________________________Global variable declarations_________________________}
   {variables used in the encoder }
.GLOBAL     u, dp, L_ACF, scaleauto;
.GLOBAL     old_LARpp, mp, L_z2_l, L_z2_h, z1;
```

(listing continues on next page)

```
   {variables used in the decoder }
.GLOBAL     nrp, drp, old_LARrpp, msr, v;

   {variables used for comfort noise insertion in the encoder}
.GLOBAL     cni_wait, speech_count, oldlar_pntr, oldxmax_pntr;
.GLOBAL     oldlar_buffer, oldxmax_buffer, sp_flag;

   {variable used as a working buffer to alias VAD variables}
.GLOBAL     wt;
{____________________________________________________________________________}

{____________________________Encoder Subroutine______________________________}
dmr_encode: ENA AR_SAT;                 {Enable ALU saturation}
         DM(speech_in)=I0;              {Save pointer to input window}
         DM(xmit_buffer)=I1;            {Save pointer to coeff window}
         MX1=H#4000;                    {This multiply will place the}
         MY1=H#100;                     {vale of H#80 in MF that will}
         MF=MX1*MY1 (SS);               {be used for unbiased rounding}

{  This section of code computes the downscaling and offset compensation
   of the input signal as described in sections 4.2.1 and 4.2.2 of the
recommendation}
         I0=DM(speech_in);              {Get pointer to input data}
         I1=I0;                         {Set pointer for output data}
         SE=-15;                        {Commonly used shift value}
         MX1=H#80;                      {Used for unbaised rounding}
         AX1=16384;                     {Used to round result}
         MY0=32735;                     {Coefficient value}
         AY1=H#7FFF;                    {Used to mask lower L_z2}
         MY1=DM(z1);
         MR0=DM(L_z2_l);
         MR1=DM(L_z2_h);
         DIS AR_SAT;                    {Cannot do saturation}
         AR=MR0 AND AY1, SI=DM(I1,M1);  {Fill the pipeline}
         CNTR=window_length;
{        DO offset_comp UNTIL CE;}
gsm1:            SR=ASHIFT SI BY -3 (HI);{Shift input data to zero the}
            SR=LSHIFT SR1 BY 2 (HI);    {the LSB and half data}
            AX0=SR1, SR=ASHIFT MR1 (HI);  {Get upper part of L_z2 (msp)}
            SR=SR OR LSHIFT MR0 (LO);   {Get LSB of L_z2 (lsp)}
            MR=MX1*MF (SS), MX0=SR0;    {Prepare MR, MX0=msp}
            MR=MR+AR*MY0 (SS), AY0=MY1;   {Compute temp}
            AR=AX0-AY0, AY0=MR1;        {Compute new s1}
            SR=ASHIFT AR BY 15 (LO);    {Compute new L_s2}
            AR=SR0+AY0, MY1=AX0;        {MY1 holds z1, L_s2+temp is in}
            AF=SR1+C, AY0=AR;           {SR in double precision}
            MR=MX0*MY0 (SS);            {Compute msp*32735}
            SR=ASHIFT MR1 BY -1 (HI);   {Downshift by one bit }
            SR=SR OR LSHIFT MR0 BY -1 (LO);{before adding to L_s2}
```

```
            AR=SR0+AY0, AY0=AX1;          {Compute new L_z2 in }
            MR0=AR, AR=SR1+AF+C;          {double precision MR0=L_z2}
            MR1=AR, AR=MR0+AY0;           {MR1=L_z2, round result }
            SR=LSHIFT AR (LO);            {and downshift for output}
            AR=MR1+C, SI=DM(I1,M1);       {Get next input sample}
            SR=SR OR ASHIFT AR (HI);
offset_comp:      DM(I0,M1)=SR0, AR=MR0 AND AY1;{Store result, get next lsp}
{?} IF NOT CE JUMP gsm1;
         DM(L_z2_l)=MR0;                        {Save values for next call}
         DM(L_z2_h)=MR1;
         DM(z1)=MY1;
         ENA AR_SAT;                            {Re-enable ALU saturation}

{  This section of code computes the pre-emphasis filter and
   the autocorrelation as defined in sections 4.2.3 and 4.2.4 of
   the recommendation}
         MX0=DM(mp);                            {Get saved value for mp}
         MY0=-28180;                            {MY0 holds coefficient value}
         MX1=H#80;                              {These are used for biased}
         MR=MX1*MF (SS);                        {rounding}
         SB=-4;                                 {Maximum scale value}
         I0=DM(speech_in);                      {In-place computation}
         CNTR=window_length;
{        DO pre_emp UNTIL CE;}
gsm2:             MR=MR+MX0*MY0 (SS), AY0=DM(I0,M0);
            AR=MR1+AY0, MX0=AY0;
            SB=EXPADJ AR;                 {Check for maximum value}
pre_emp:    DM(I0,M1)=AR, MR=MX1*MF (SS); {Save filtered data}
{?} IF NOT CE JUMP gsm2;
         DM(mp)=MX0;

         AY0=SB;                          {Get exponent of max value}
         AX0=4;                           {Add 4 to get scale value}
         AR=AX0+AY0;
         DM(scaleauto)=AR;                {Save scale for later}
         IF LE JUMP auto_corr;            {If 0 scale, only copy data}
         AF=PASS 1;
         AR=AF-AR;
         SI=16384;
         SE=AR;
         I0=DM(speech_in);
         I1=I0;                           {Output writes over the input}
         SR=ASHIFT SI (HI);
         AF=PASS AR, AR=SR1;              {SR1 holds temp for multiply}
         MX1=H#80;                        {Used for unbiased rounding}
         MR=MX1*MF (SS), MY0=DM(I0,M1);   {Fetch first value}
CNTR=window_length;
```

(listing continues on next page)

```
{        DO scale UNTIL CE;}
gsm3:            MR=MR+SR1*MY0 (SS), MY0=DM(I0,M1);  {Compute scaled data}
scale:      DM(I1,M1)=MR1, MR=MX1*MF (SS);           {Save scaled data}
{?} IF NOT CE JUMP gsm3;

auto_corr:  I1=DM(speech_in);               {This section of code computes}
         I5=I1;                             {the autocorr section for LPC}
         I2=window_length;                  {I2 used as down counter}
         I6=^L_ACF;                         {Set pointer to output array}
         CNTR=9;                            {Compute nine terms}
{        DO corr_loop UNTIL CE;}
gsm4:            I0=I1;                     {Reset pointers for mac loop}
            I4=I5;
            MR=0, MX0=DM(I0,M1);            {Get first sample}
            CNTR=I2;                        {I2 decrements once each loop}
{           DO data_loop UNTIL CE;}
gsm5:                  MY0=DM(I4,M5);
data_loop:             MR=MR+MX0*MY0 (SS), MX0=DM(I0,M1);
{?} IF NOT CE JUMP gsm5;
            MODIFY(I2,M2);                  {Decrement I2, Increment I5}
            MY0=DM(I5,M5);
            DM(I6,M5)=MR1;                  {Save double precision result}
corr_loop:       DM(I6,M5)=MR0;             {MSW first}
{?} IF NOT CE JUMP gsm4;

         I0=DM(speech_in);                  {This section of code rescales}
         SE=DM(scaleauto);                  {the input data}
         I1=I0;                             {Output writes over input}
         SI=DM(I0,M1);
         CNTR=window_length;
{        DO rescale UNTIL CE;}
gsm6:            SR=ASHIFT SI (HI), SI=DM(I0,M1);
rescale:    DM(I1,M1)=SR1;
{?} IF NOT CE JUMP gsm6;

         call vad_routine;                  {determine vad state}

{*****  This section of code sets the Voice Activity Flag (vad) and, if
     vad has been inactive four or more cycles (cni_wait), sets the
     Comfort Noise Insert Flag (cni_flag).  *****}
set_flags:  AX0 = DM(vad);                  {AX0 holds vad}

{__________________________Conditional Assembly______________________________}
{  Use (asm21 -cp -Ddemo) to turn on the demonstration functions}
#ifdef demo
set_vad_demo:AY0 = 2;
         MR0 = M7;
         AF  = PASS 1;
         AR  = MR0 AND AF;                  {extract force_vad_low}
         IF NE AF = PASS 0;
         AR  = AX0 AND AF;                  {AR = vad AND /force_vad_low }
```

```
        AF  = MR0 AND AY0;                  {extract force_vad_high}
        AR  = AR OR AF;                     {AR = .. OR force_vad_high }
        DM(vad) = AR;
        AX0 = AR;
        M7  = 2;
#endif
{_____________________________________________________________________________}

{___________________________Conditional Assembly______________________________}
{    Use (asm21 -cp -Dnovad) to turn VAD off for validation            }
#ifdef novad
        AX0 = 1;
        DM(vad) = AX0;
#endif
{_____________________________________________________________________________}

        AY0 = DM(cni_wait);
        AY1 = DM(speech_count);

        MR0 = H#FFFF;                       {MR0 holds cni_flag}
        AR  = -4;                           {AR holds cni_wait}

        AF  = PASS AX0;
        IF NE MR = 0;                       {If vad<>0, set cni_flag=0}
        IF NE JUMP store_cni;

        AR  = AY0 + 1;                      {Increment cni_wait}
        IF LE MR = 0;                       {If cni_wait <= 0, cni_flag=0}

store_cni:  DM(cni_wait) = AR;
        DM(cni_flag)  = MR0;

        AY0 = -24;
        AF  = PASS MR0;
        IF NE AR = PASS AY0;
        IF NE JUMP store_spcnt;

        AF  = PASS AX0, AR = AY1;
        IF NE AR = AY1 + 1;
store_spcnt:DM(speech_count) = AR;

        AF  = PASS AX0, AY1 = AR; AR  = 0;
        IF NE AR = PASS 1;
        AF = PASS AY1;
        IF GE AR = PASS 1;
store_spflg:DM(sp_flag) = AR;
```

(listing continues on next page)

```
{  Now begin section 4.2.5 of the recommendation}
set_up_schur:AY1 = ^L_ACF;          {in DM}
         MY1 = ^acf;
         M0  = ^r;
         CALL schur_routine;

{  This section of code transforms the r-values to log-area-ratios
   as defined in section 4.2.6 of the recommendation}

real_rs: I5=^r;                     {This section of code computes}
         I4=^LAR;                   {the log area ratio from r}
         CNTR=8;
{        DO compute_lar UNTIL CE;   }
gsm7:             AX0=DM(I5,M5);
            AR=ABS AX0;
            SR=ASHIFT AR BY -1 (HI);{Generate temp>>1}
            AX0=SR1;                {AX0 holds temp>>1}
            AY0=26112;
            AX1=AR, AR=AR-AY0;      {Generate temp-26112}
            SR=LSHIFT AR BY 2 (HI); {Generate (temp-26112)<<2}
            AY0=31130;
            AY1=11059;
            AR=SR1, AF=AX1-AY0;     {Default to AR=(temp-26112)<<2}
            IF LT AR=AX1-AY1;       {AR=temp-11059 (if necessary)}
            AY0=22118;
            AF=AX1-AY0;
            IF LT AR=PASS AX0;      {AR=temp>>1 (if necessary)}
            IF NEG AR=-AR;          {Compute sign of LAR[i]}
compute_lar:      DM(I4,M5)=AR;           {Save LAR[i]}
{?} IF NOT CE JUMP gsm7;

{*****  If necessary, the code will now average the LAR values, and write
     new values into oldlar_buffer.  The proper LAR values are then
     transmitted. *****}

         AX0 = DM(vad);
         AF  = PASS AX0;
         IF NE JUMP encode_lar;     {Voice Activity, skip the rest}
         AX0 = DM(cni_flag);
         AF  = PASS AX0;
         IF EQ JUMP write_oldlar;   {Not cni, so do not avg. oldlar}

{*****  The code will now average the four previous frames lar values as
     specified in GSM recommendation 06.12.  Note that the values were
     previously scaled. *****}

         I4  = ^oldlar_buffer;
         I5  = ^mean_lar;
         I6  = I4;
         M7  = 8;
         AX0 = DM(I6,M7);
         CNTR = 7;
```

```
{        DO average_lar UNTIL CE;}
gsm8:             MODIFY (I4,M5);
            AY0 = DM(I6,M7);
            AF  = AX0 + AY0, AX0 = DM(I6,M7);
            AF  = AX0 + AF, AX0 = DM(I6,M7);
            I6  = I4;
            AR  = AX0 + AF, AX0 = DM(I6,M7);
average_lar:      DM(I5,M5) = AR;         {store mean_lar[i]}
{?} IF NOT CE JUMP gsm8;
         AY0 = DM(I6,M7);
         AF  = AX0 + AY0, AX0 = DM(I6,M7);
         AF  = AX0 + AF, AX0 = DM(I6,M7);
         AR  = AX0 + AF;
         DM(I5,M5) = AR;                  {store mean_lar[8]}
         M7  = 2;                         {restore M7}

{*****  This section of code will write the current lar values into one
     of four (eight location) buffers in the thirty-two location
     oldlar_buffer for use in the next frame.  The values are also
     scaled. *****}

write_oldlar:AX0 = ^oldlar_buffer;
         AY1 = ^oldlar_buffer + 32;
         AR  = DM(oldlar_pntr);
         AF  = AY1 - AR;
         IF LE AR = PASS AX0;
         I4  = AR;                        {Set the top of buffer}
         SE  = -2;                        {Roughly divide by four}
         I5  = ^LAR;
         SI  = DM(I5,M5);
         CNTR = 8;
{        DO write_buffer UNTIL CE;}
gsm9:             SR = ASHIFT SI (HI), SI = DM(I5,M5);   {last read will be junk}
write_buffer:     DM(I4,M5) = SR1;
{?} IF NOT CE JUMP gsm9;
         DM(oldlar_pntr) = I4;

{*****  This code will quantize the current LAR values and the mean_lar values, if
necessary.  One of these is then sent to the transmit buffer. *****}

encode_lar: I6  = ^LAR;
         I1  = ^LARc;
         CALL lar_encoding;

         AX0 = DM(sp_flag);
         AF = PASS AX0;
         IF NE JUMP transmit_lar;

         I6  = ^mean_lar;
         I1  = ^mean_larc;
         CALL lar_encoding;
```

(listing continues on next page)

```
transmit_lar:     I1 = AX1;                {The quantized LAR values}
         CNTR=8;                           {can now be sent}
         CALL xmit_data;                   {Copy to the output buffer}

{  Now, continue with GSM recommendation 4.2.8.}

         CALL decode_larc;                 {Decode the LARcs }
         I0=DM(speech_in);                 {Input/output of the st filter}
         I6=^st_analysis;                  {Use the st analysis routine}
         I4=^old_larpp;                    {Use the previous LARpp}
         CALL st_filter;                   {Call st filter manager}

{  Compute sub-window information for each of the 4 sub-windows}

{*****  Check to see if Comfort Noise is being generated. *****}

         AX0 = DM(sp_flag);
         AF  = PASS AX0;
         IF NE JUMP speech_frame;

         AX0 = DM(cni_flag);
         AF  = PASS AX0;
         IF NE JUMP comp_mnxmax;

silence_frame:AR  = DM(mean_xmaxc);
         JUMP xmit_cmfrtnois;

{*****  This section will average the four xmax values from the previous
four frames as specified in GSM recommendation 06.12, section 2.1. Note
that the values have been pre-scaled. *****}

comp_mnxmax:I5  = ^oldxmax_buffer;
         AR  = DM(I5,M5);                  {AR holds mean_xmax.}
         AY0 = DM(I5,M5);
         CNTR = 15;
{        DO avg_xmax UNTIL CE;}
avg_xmax:         AR = AR + AY0, AY0 = DM(I5,M5); {Last read is junk.}
{?} IF NOT CE JUMP avg_xmax;


{*****  Now xmax must be quantized. *****}

         CALL quantize_xmax;               {mean_xmaxc returned in AR.}
         DM(mean_xmaxc) = AR;

{*****  The transmit buffer is filled next. *****}
xmit_cmfrtnois:CNTR = 4;
         AX0 = 0;
         I0  = DM(xmit_buffer);
{        DO xmit_sid UNTIL CE;}
```

```
gsm10:      DM(I0,M1) = AX0;
            DM(I0,M1) = AX0;
            DM(I0,M1) = AX0;
            DM(I0,M1) = AR;         {The fourth value is mean_xmaxc}
            CNTR = 12;
{           DO zero_rpe UNTIL CE;}
zero_rpe:               DM(I0,M1) = AX0;
{?} IF NOT CE JUMP zero_rpe;
xmit_sid:         DM(I0,M1) = AX0;
{?} IF NOT CE JUMP gsm10;

{*****  The Silence Descriptor (SID) frame has been sent to the transmit
     buffer. *****}

{*****  Must now compute the xmax values for the current frame. *****}

         I3  = DM(speech_in);
         I6=^lags;

         CNTR=4;
{        DO xmax_loop UNTIL CE;}
gsm11:      CALL ltp_computation;
            DM(I6,M5) = AX1;        {AX1 holds Nc for sub-window}
            CALL rpe_encoding;
xmax_loop:        NOP;
{?} IF NOT CE JUMP gsm11;
         JUMP finish;

{  This code implements the sub-window information for each of the 4
   speech sub-windows.}

speech_frame:I3=DM(speech_in);      {Only set input pointer once}
         I6=^lags;

         CNTR=4;
{        DO enc_subwindow UNTIL CE;}
gsm12:      CALL ltp_computation;   {Compute LTP coefficients}
            DM(I6,M5) = AX1;        {AX1 holds Nc for sub-window}
            CALL rpe_encoding;      {Encode and decode RPE sequence}
                  I1=^Nc;           {Sub-window data can be sent}
                  CNTR=17;          {17 coeffs per sub-window}
                  CALL xmit_data;   {Copy to the output buffer}
enc_subwindow:          NOP;        {No CALL in last instr of DO}
{?} IF NOT CE JUMP gsm12;

   {All the coded variables have been sent to xmit_buffer}

finish:  CALL update_periodicity;   {VAD (06.32) routine}

         DIS AR_SAT;
         RTS;                       {Return to caller}
```

(listing continues on next page)

```
xmit_data:  I0=DM(xmit_buffer);      {Copy coeffs to the output}
{         DO xmit UNTIL CE;}         {buffer}
gsm13:      AX0=DM(I1,M1);
xmit:             DM(I0,M1)=AX0;
{?} IF NOT CE JUMP gsm13;
          DM(xmit_buffer)=I0;
          RTS;                       {Return from Encoder}

{__________________________Subroutines for Encoder___________________________}

{  This section of code quantizes and codes the LAR value produced above
   as defined in section 4.2.7 of the recommendation}

lar_encoding:AX1 = I1;               {Stores pointer to result}
          I5=^table_a;               {This section of code computes}
          I4=^table_b;               {the quantizing/coding of LARs}
          MX1=^table_mac;            {Pointers are set to various}
          MY1=^table_mic;            {data memory tables}
          AX0=256;                   {Used for rounding}
          CNTR=8;
{         DO quantize_lar UNTIL CE;}
gsm14:      MX0=PM(I5,M5);
            SI=I5;
            MY0=DM(I6,M5);
            MR=MX0*MY0 (SS), AY0=PM(I4,M5);     {temp=A[i]*LAR[i]}
            AF=MR1+AY0;                         {temp=A[i]*LAR[i]+B[i]}
            I5=MX1;
            AR=AX0+AF, AY0=PM(I5,M5);           {Round result}
            MX1=I5;
            SR=ASHIFT AR BY -9 (HI);            {LARc[i] = temp>>9}
            AR=SR1;
            I5=MY1;
            AF=AR-AY0, AY1=PM(I5,M5);           {Test min/max}
            MY1=I5;
            IF GT AR=PASS AY0;                  {Cap if above max}
            AF=AR-AY1;
            IF LT AR=PASS AY1;                  {of below min}
            AR=AR-AY1;                          {Subtract minimum value}
            I5=SI;
quantize_lar:     DM(I1,M1)=AR;                 {Save LARc[i]}
{?} IF NOT CE JUMP gsm14;
          RTS;
```

```
{  This subroutine computes the 8-pole short term lattice filter
   as defined in section 4.2.10 of the recommendation}

st_analysis:SR1=H#80;                     {Used for unbaised rounding}
{        DO st_compute UNTIL CE;}         {The counter is set by caller}
gsm15:      I5=^rp;                       {Point to decoded r-values}
            I2=^u;                        {Point to delay line}
            AR=DM(I0,M0);                 {Get filter input}
            AX0=AR;                       {Set sav=s[i], AX0 is sav}
            CNTR=8;                       {Compute all 8 poles}
{           DO st_loop UNTIL CE;}
gsm16:            MY0=DM(I5,M5);                      {Moved to dm}
                  MR=SR1*MF (SS), MX1=DM(I2,M0);
                  MR=MR+AR*MY0 (SS), AY0=MX1;
                  AY1=AR, AR=MR1+AY0;                 {AR=temp}
                  DM(I2,M1)=AX0, MR=SR1*MF (SS);      {u[i-1]=sav}
                  MR=MR+MX1*MY0 (SS);
st_loop:          AX0=AR, AR=MR1+AY1;                 {AR=di, AX0=sav}
{?} IF NOT CE JUMP gsm16;
st_compute:       DM(I0,M1)=AR;                       {Write output over input}
{?} IF NOT CE JUMP gsm15;
         RTS;

{  This section of code computes the maximum cross-correlation value
   of the reconstructed short term signal dp() and the current
   sub-window as defined in section 4.2.11 of the recommendation}

ltp_computation:I0=I3;                    {Preserve I3 for now}
         SB=-6;                           {Maximum shift value}
         SI=DM(I0,M1);
         CNTR=sub_window_length;
{        DO find_dmax UNTIL CE;}
find_dmax:        SB=EXPADJ SI, SI=DM(I0,M1);{Find maximum of sub-window}
{?} IF NOT CE JUMP find_dmax;

         AY0=6;
         AX0=SB;
         AR=AX0+AY0;                      {Compute shift for scaling}
         DM(scal)=AR;                     {Save shift value}
         AR=-AR;
         SE=AR;
         I1=^wt;                          {Output to temporary array}
         I0=I3;                           {Preserve I3 for now}
         SI=DM(I0,M1);
         CNTR=sub_window_length;          {Scale entire sub-window}
{        DO init_wt UNTIL CE;}
gsm17:      SR=ASHIFT SI (HI), SI=DM(I0,M1);
init_wt:    DM(I1,M1)=SR1;
{?} IF NOT CE JUMP gsm17;
```

(listing continues on next page)

```
        DIS AR_SAT;                         {Can use saturation here}
        AX1=40;                             {Mimimum value for Nc}
        I0=39;                              {I0 holds Nc counter}
        AY0=0;                              {Holds LSW of max value}
        AY1=0;                              {Holds MSW of max value}
        I4=^dp+80;
        I2=^wt;
        I1=I2;
        CNTR=81;
{       DO cross_loop UNTIL CE;}
gsm18:     I5=I4;
           MR=0, MX0=DM(I1,M1), MY0=PM(I5,M5);
           CNTR=sub_window_length;
{          DO cross_corr UNTIL CE;}
cross_corr:            MR=MR+MX0*MY0 (SS), MX0=DM(I1,M1), MY0=PM(I5,M5);
{?} IF NOT CE JUMP cross_corr;
           AR=MR0-AY0, MY0=PM(I4,M6);
           AR=MR1-AY1+C-1;
           MODIFY(I0,M1);
           IF LT JUMP cross_loop;           {Check for L_result < L_max}
           IF EQ AR=MR0-AY0;                {If MSW=0, check LSW again}
           IF EQ JUMP cross_loop;           {If LSW=0, the values are equal}
           AY0=MR0;                         {Reset L_MAX to new value}
           AY1=MR1;                         {in double precision}
           AX1=I0;                          {AX1 holds current value for Nc}
cross_loop:      I1=I2;                     {Reset pointer into array}
{?} IF NOT CE JUMP gsm18;
        DM(Nc)=AX1;                         {After loop, Nc is in AX1}

        SI=AY1;                             {This section of code computes}
        AY1=6;                              {the power of the reconstructed}
        AX0=DM(scal);                       {short term residual signal dp}
        AR=AX0-AY1;
        SE=AR;
        SR=ASHIFT SI (HI), AR=AY0;
        SR=SR OR LSHIFT AR (LO);
        SE=-3;
        AY0=^dp+120;                        {Use dp() array directly, do}
        AR=AY0-AX1, AY0=SR0;                {not bother with temp array}
        AY1=SR1;
        I5=AR;
        MR=0, AR=PM(I5,M5);
        CNTR=sub_window_length;
{       DO power UNTIL CE;}
gsm19:     SR=ASHIFT AR (HI), AR=PM(I5,M5);       {Scale data}
           MY0=SR1;                               {Copy to y-reg}
power:     MR=MR+SR1*MY0 (SS);                    {Compute L_power}
{?} IF NOT CE JUMP gsm19;
```

```
        AR=0;                        {This section of code computes}
        AF=PASS AY1;                 {and codes the LTP gain value}
        IF LT JUMP bc_found;         {L_max < 0, so bc=0}
        IF EQ AF=PASS AY0;
        IF EQ JUMP bc_found;         {L_max = 0, so bc=0}
        AR=3;
        AF=MR0-AY0;
        AF=MR1-AY1+C-1;
        IF LT JUMP bc_found;         {L_max > L_power, so bc=3}
        IF EQ AF=MR0-AY0;
        IF EQ JUMP bc_found;         {L_max = L_power, so bc=3}
        SE=EXP MR1 (HI);             {Normalize L_power}
        SE=EXP MR0 (LO)
        SR=NORM MR1 (HI), MR1=AY1;
        SR=SR OR NORM MR0 (LO), MR0=AY0;
        MY0=SR1, SR=NORM MR1 (HI); {Normalize L_max, MY0 holds s}
        SR=SR OR NORM MR0 (LO);
        AY0=SR1, AF=PASS 0;          {AY0 holds R}
        I5=^table_dlb;               {Check for each value of bc}
        AR=PASS 0, MX0=PM(I5,M5);
        MR=MX0*MY0 (SS), MX0=PM(I5,M5);
        AF=MR1-AY0;
        IF GE JUMP bc_found;
        AR=1;
        MR=MX0*MY0 (SS), MX0=PM(I5,M5);
        AF=MR1-AY0;
        IF GE JUMP bc_found;
        AR=2;
        MR=MX0*MY0 (SS);
        AF=MR1-AY0;
        IF GE JUMP bc_found;
        AR=3;

bc_found:   DM(bc)=AR;               {AR holds the value of bc}
        ENA AR_SAT;                  {Re-enable ALU saturation}

{  This section of code computes the long term analysis filtering section
   as described in section 4.2.12 of the recommendation}

lt_analysis:AY0=^table_qlb;
        AR=AR+AY0;
        I5=AR;
        MY0=PM(I5,M4);
        AY0=^dp+120;
        AR=AY0-AX1;
        I4=AR;
        I5=^dpp;                     {Output array dpp()}
        I2=^wt+5;                    {The e-array goes into wt}
        MX1=H#80;
        MR=MX1*MF (SS), MX0=PM(I4,M5);
```

(listing continues on next page)

```
        CNTR=sub_window_length;
{       DO calculate_e UNTIL CE;}
gsm20:     MR=MR+MX0*MY0 (SS), AY0=DM(I3,M1);  {Compute dpp[k]}
           AR=AY0-MR1, MX0=PM(I4,M5);          {Compute e[k]}
           DM(I5,M5)=MR1;                      {Save dpp()}
calculate_e:     DM(I2,M1)=AR, MR=MX1*MF (SS); {Save e() into wt()}
{?} IF NOT CE JUMP gsm20;

{  All the long term parameters (Nc, bc, mc) have been computed}

        RTS;

{  This subroutine computes, encodes and decodes the Residual Pulse
   Excitation sequence as defined in section 4.2.13 of the recommendation}

rpe_encoding:I0=^wt;                     {The beginning of wt must be}
        AX0=0;                           {cleared for use in the block}
        CNTR=5;                          {filter}
{       DO zero_start UNTIL CE;}
zero_start:      DM(I0,M1)=AX0;
{?} IF NOT CE JUMP zero_start;
        I0=^wt+45;                       {The end must also be cleared}
        CNTR=5;
{       DO zero_end UNTIL CE;}
zero_end:        DM(I0,M1)=AX0;
{?} IF NOT CE JUMP zero_end;

        DIS AR_SAT;
        I2=^wt;                          {wt will be reloaded with x()}
        CNTR=sub_window_length;
{       DO compute_x_array UNTIL CE;}
gsm21:     I0=I2;
           I4=^h;
           MR=0, MX0=DM(I0,M1), MY0=PM(I4,M5);
           MR0=8192;                     {Used for rounding}
           CNTR=11;                      {11-term filter}
{          DO compute_x UNTIL CE;}
compute_x:             MR=MR+MX0*MY0 (SS), MX0=DM(I0,M1), MY0=PM(I4,M5);
{?} IF NOT CE JUMP compute_x;
           AY0=MR0;                      {The output value must be}
           AR=MR0+AY0, AY0=MR1;          {Up-shifted with saturation}
           AY1=AR, AR=MR1+AY0+C;
           IF NOT AV JUMP done_2x;       {Check for overflow on 2x}
           AR=H#7FFF;                    {Overflow. manually saturate}
           AY1=H#8000;                   {output, and save value}
           IF AC AR=PASS AY1;
           JUMP compute_x_array;
done_2x:   AX1=AR, AR=PASS AY1;          {Compute 4x}
           AY0=AX1, AR=AR+AY1;
           ENA AR_SAT;                   {Automatic saturation can}
           AR=AX1+AY0+C;                 {be used on the last add}
           DIS AR_SAT;
```

```
compute_x_array:  DM(I2,M1)=AR;           {Output writes over input}
{?} IF NOT CE JUMP gsm21;

{  This section of code computes the RPE Grid Selection as described
   in section 4.2.14 of the recommendation}

         AF=PASS 0;
         AY0=0;
         AY1=0;
         AX0=0;
         M0=3;                            {Used for interleaving}
         I1=^wt;
         CNTR=4;
{        DO find_mc UNTIL CE;}
gsm22:      I2=I1;
            MR=0, SI=DM(I2,M0);           {L_result=0, fetch first value}
            CNTR=13;
{           DO calculate_em UNTIL CE;}
gsm23:            SR=ASHIFT SI BY -2 (HI);{Downshift to avoid overflow}
                  MY0=SR1;                {Copy to yop}
calculate_em:           MR=MR+SR1*MY0 (SS), SI=DM(I2,M0);{L_result is in MR}
{?} IF NOT CE JUMP gsm23;
            AR=MR0-AY0;
            AR=MR1-AY1+C-1;
            IF LT JUMP find_mc;           {Check for L_result<EM}
            IF EQ AR=MR0-AY0;             {If MSW=0, check LSW again}
            IF EQ JUMP find_mc;           {L_result = EM}
            AY0=MR0;                      {EM=L_result}
            AY1=MR1, AR=PASS AF;
            AX0=AR;                       {Mc=m}
find_mc:    AF=AF+1, MX0=DM(I1,M1);
{?} IF NOT CE JUMP gsm22;
                                          {Mc in AX0}
         ENA AR_SAT;
         DM(Mc)=AX0;
         AY0=^wt;
         I1=^wt;                          {temp array will be reloaded}
         AR=AX0+AY0;                      {with xM()}
         I0=AR;
         AR=PASS 0;
         CNTR=13;
{        DO decimate UNTIL CE;}
gsm24:      AX0=DM(I0,M0);                {Read every third value}
            AF=ABS AX0, DM(I1,M1)=AX0;    {Check for maximum value}
            AF=AR-AF;
decimate:         IF LT AR=ABS AX0;       {AR holds xmax}
{?} IF NOT CE JUMP gsm24;
         M0=0;                            {Reset M0 to usual value}
```

(listing continues on next page)

```
{*****  The following code checks vad and stores xmax in oldxmax_buffer,
   if necessary. *****}

         AX0 = DM(vad);
         AF  = PASS AX0;
         IF NE JUMP xmax_speech;                {Yes - VAD, so do not store}
         SI  = AR;                              {Save xmax in SI}

{*****  This section of code will write xmax into the oldxmax_buffer for use
in the next frame.  Note that scaling also takes place. *****}

         AX0 = ^oldxmax_buffer;
         AY1 = ^oldxmax_buffer + 16;
         AR  = DM(oldxmax_pntr);
         AF  = AY1 - AR;
         IF LE AR = PASS AX0;                   {AR holds address}

         SR  = ASHIFT SI BY -4 (HI);            {SR1 holds scaled xmax}
         I5  = AR;
         DM(I5,M5) = SR1;                       {Write xmax to oldxmax_buffer}
         DM(oldxmax_pntr) = I5;
         AR  = SI;                              {Restore xmax}

{  This section of code computes the APCM quantization of the selected
RPE section as defined in section 4.2.15 of the recommendation}

xmax_speech:CALL quantize_xmax;     {input and output in AR}
         DM(xmaxc)=AR;              {Save xmaxc}
         CALL get_xmaxc_pts;        {Compute exponent and mantissa}
                                    {Exponent in AY1}
         AY0=^table_nrfac;          {Mant in AR}
         MX0=AR, AR=AR+AY0;         {Now mant in MX0}
         I5=AR;
         MY0=PM(I5,M5);             {MYO holds temp2}
         AX0=6;
         AR=AX0-AY1;
         AY0=4;
         I0=^wt;                    {Temp array current holds xM()}
         I2=^xmc;
         SE=AR;                     {SE holds temp1}
         SI=DM(I0,M1);
         CNTR=13;
{        DO compute_xm UNTIL CE;}
gsm25:      SR=LSHIFT SI (HI), SI=DM(I0,M1);    {temp=xM[i]<<temp1}
            MR=SR1*MY0 (SS);                    {temp=temp*temp2}
            SR=ASHIFT MR1 BY -12 (HI);
            AR=SR1+AY0; {AR=xMc[i]}
compute_xm:       DM(I2,M1)=AR;                 {Store xMc[i]}
{?} IF NOT CE JUMP gsm25;

         CALL rpe_decoding;                     {APCM inverse quantization}
```

```
{  This section of code updates the reconstructed short term residual
   signal dp() as defined in section 4.2.18 of the recommendation}

update_dp_code: I4=^dp;                         {I4 points to dp[-120]}
         I5=^dp+40;                             {I5 points to dp[-80]}
         CNTR=80;
{        DO update_dp UNTIL CE;     }
gsm26:      AX0=PM(I5,M5);
update_dp:        PM(I4,M5)=AX0;                {dp[-120+k]=dp[-80+k]}
{?} IF NOT CE JUMP gsm26;
         I4=^dp+80;                             {I4 points to dp[-40]}
         I1=^ep;
         I5=^dpp;
         AX0=DM(I1,M1);
         AY0=DM(I5,M5);                         {Fetch first samples}
         CNTR=sub_window_length;
{        DO fill_dp UNTIL CE;}
gsm27:      AR=AX0+AY0, AX0=DM(I1,M1);
            AY0=DM(I5,M5);
fill_dp:    PM(I4,M5)=AR;                       {dp[-40+k]=ep[k]+dpp[k]}
{?} IF NOT CE JUMP gsm27;
         RTS;

{  This section of code computes the APCM quantization of the selected
   RPE section as defined in section 4.2.15 of the recommendation}

quantize_xmax:SI=AR, AF=PASS 0;           {This section of code quantizes}
         SR=ASHIFT AR BY -9 (HI);         {and codes xmax into xmaxc}
         CNTR=6;
{        DO get_exp UNTIL CE;}
gsm28:      AR=PASS SR1;                  {SR1 holds temp}
            IF GT AF=AF+1;                {Increment exp until SR1=0}
get_exp:    SR=ASHIFT SR1 BY -1 (HI);
{?} IF NOT CE JUMP gsm28;

         AX1=5;
         AR=AX1+AF;                       {temp=exp+5}
         AR=-AR;                          {Use this for downshift}
         SE=AR, AR=PASS AF;               {AR=exp}
         SR=LSHIFT AR BY 3 (HI);          {Place exponent}
         AY0=SR1, SR=ASHIFT SI (HI);      {Place mantissa}
         AR=SR1+AY0;                      {AR holds xmaxc}
         RTS;
```

(listing continues on next page)

4 GSM Codec

```
{______________Encoder and Voice Activity Detector Subroutines_______________}

{  This section of code computes the reflection coefficients using the
   schur recursion as defined in section 4.2.5 of recommendation 6.10 and
   section 3.3.1 of recommendation 6.32}

schur_routine:I6=AY1;                    {This section of code prepares}
         AR=DM(I6,M5);                   {for the schur recursion}
         SE=EXP AR (HI), SI=DM(I6,M5);   {Normalize the autocorrelation}
         SE=EXP SI (LO);                 {sequence based on L_ACF[0]}
         SR=NORM AR (HI);
         SR=SR OR NORM SI (LO);
         AR=PASS SR1;                    {If L_ACF[0] = 0, set r to 0}
         IF EQ JUMP zero_reflec;
         I6=AY1;
         I5=MY1;
         AR=DM(I6,M5);
         CNTR=9;                         {Normalize all terms}
{        DO set_acf UNTIL CE;}
gsm29:      SR=NORM AR (HI), AR=DM(I6,M5);
            SR=SR OR NORM AR (LO), AR=DM(I6,M5);
set_acf:    DM(I5,M5)=SR1;
{?} IF NOT CE JUMP gsm29;

         I5=MY1;                         {This section of code creates}
         I4=^k+7;                        {the k-values and p-values}
         I0=^p;
         AR=DM(I5,M5);                   {Set P[0]=acf[0]}
         DM(I0,M1)=AR;
         CNTR=7;
{        DO create_k UNTIL CE;}          {Fill the k and p arrays}
gsm30:      AR=DM(I5,M5);
            DM(I0,M1)=AR;
create_k:         DM(I4,M6)=AR;
{?} IF NOT CE JUMP gsm30;
         AR=DM(I5,M5);
         DM(I0,M1)=AR;                   {Set P[8]=acf[8]}

         I5=M0;                          {Compute r-values}
         I6=7;                           {I6 used as downcounter}
         SR0=0;
         SR1=H#80;                       {Used in unbiased rounding}
         CNTR=7;                         {Loop through first 7 r-values}
{        DO compute_reflec UNTIL CE;}
gsm31:      I2=^p;                       {Reset pointers}
            I4=^k+7;
            AX0=DM(I2,M1);               {Fetch P[0]}
            AX1=DM(I2,M2);               {Fetch P[1]}
            MX0=AX1, AF=ABS AX1;         {AF=abs(P[1])}
            AR=AF-AX0;
```

```
            IF LE JUMP do_division;       {If P[0]<abs(P[1]), r = 0}
            DM(I5,M5)=SR0;                {Final r =0}
            JUMP compute_reflec;
do_division:      CALL divide_routine;    {Compute r[n]=abs(P[1])/P[0]}
            AR=AY0, AF=ABS AX1;
            AY0=32767;
            AF=AF-AX0;                    {Check for abs(P[1])=P[0]}
            IF EQ AR=PASS AY0;            {Saturate if they are equal}
            IF POS AR=-AR;                {Generate sign of r[n]}
            DM(I5,M5)=AR;                 {Store r[n]}
            MY0=AR, MR=SR1*MF (SS);
            MR=MR+MX0*MY0 (SS), AY0=AX0;  {Compute new P[0]} AR=MR1+AY0;
            DM(I2,M3)=AR;                 {Store new P[0]}
            CNTR=I6;                      {One less loop each iteration}
{           DO schur_recur UNTIL CE;}
gsm32:            MR=SR1*MF (SS), MX0=DM(I4,M4);
                  MR=MR+MX0*MY0 (SS), AY0=DM(I2,M2);
                  AR=MR1+AY0, MX1=AY0;    {AR=new P[m]}
                  MR=SR1*MF (SS);
                  MR=MR+MX1*MY0 (SS), AY0=MX0;
                  DM(I2,M3)=AR, AR=MR1+AY0;{Store P[m], AR=new K[9-m]}
schur_recur:            DM(I4,M6)=AR;     {Store new K[9-m]}
{?} IF NOT CE JUMP gsm32;
compute_reflec:         MODIFY(I6,M6);    {Decrement loop counter (I6)}
{?} IF NOT CE JUMP gsm31;
         I2=^p;                           {Compute r[8] outside of loop}
         AX0=DM(I2,M1);                   {Using same procedure as above}
         AX1=DM(I2,M2);
         AF=ABS AX1;
         CALL divide_routine;
         AR=AY0, AF=ABS AX1;
         AY0=32767;
         AF=AF-AX0;
         IF EQ AR=PASS AY0;
         AF=ABS AX1;
         AF=AF-AX0;                       {The test for valid r is here}
         IF GT AR=PASS 0;                 {r[8]=0 if P[0]<abs(P[1])}
         IF POS AR=-AR;
         DM(I5,M5)=AR;
         JUMP schur_done;

zero_reflec:AX0=0;                        {The r-values must be set to}
         I5=M0;                           {0 according to the recursion}
         CNTR=8;
{        DO zero_rs UNTIL CE;}
zero_rs:    DM(I5,M5)=AX0;
{?} IF NOT CE JUMP zero_rs;

schur_done: M0 = 0;
         RTS;
```

(listing continues on next page)

```
{___________________________Divide Subroutine__________________________________}
divide_routine:
         AY0=0;
         DIVS AF,AX0;
         CNTR=15;
{        DO div_loop UNTIL CE;}
div_loop:        DIVQ AX0;
{?} IF NOT CE JUMP div_loop;
         RTS;
{___________________________Decoder Subroutine_________________________________}

{  This section of code implements the LPC-LTP-RPE decoder as defined in
   the GSM recommendation.}

dmr_decode: ENA AR_SAT;             {Enable ALU saturation mode}
         DM(recv_buffer)=I1;        {Save pointer to input coeff array}
         DM(speech_out)=I2;         {Save pointer to output speech array}
         MX1=H#4000;                {This is used to set the MF register}
         MY1=H#100;                 {to H#80 so that it can be used in }
         MF=MX1*MY1 (SS);           {unbiased rounding in various places}

{*****  The code will now implement the comfort noise insertion as specified
     in GSM specification 6.31, section 3.1. *****}

         AR  = PASS AX0;            {AX0 holds the SID signal}
         IF EQ JUMP start_dcd;
         CALL comfort_noise_generator;

{  Now, continue}

start_dcd:  I1=^LARc;               {Copy the LARc array into proper place}
         CNTR=8;                    {there are 8 LARcs}
         CALL recv_data;            {This subroutine copies from input buff}


         CALL decode_LARc;          {Decode the LARcs to LARs}

         I3=DM(speech_out);         {Only set output pointer once!}

         CNTR=4;                    {Computations for 4 sub windows}
{        DO dcd_subwindow UNTIL CE;}
gsm33:      I1=^Nc;                 {Set pointer to start of sub-window}
            CNTR=17;                {data array 17 coefs per sub-window}
            CALL recv_data;         {Copy them from the input buffer}
            CALL get_xmaxc_pts;     {Decode xmaxc into exp and mantissa}
            CALL rpe_decoding;      {Decode xMc array into ep array}
            CALL lt_predictor;      {Compute drp for sub-window}
            CALL setup_wtr;         {Copy drp values in temp wtr}
dcd_subwindow:    NOP;              {No CALL in last instr of DO loop}
{?} IF NOT CE JUMP gsm33;
```

```
        I0=DM(speech_out);              {Set pointer to output array}
        I1=I0;                          {Set pointer to input/output}
        I6=^st_synthesis;               {Set pointer to st filter}
        I4=^old_LARrpp;                 {Set pointer to old LARrpp}
        CALL st_filter;                 {Call short term filter manager}

{    4.3.5                                                      }

        I0=DM(speech_out);              {This section of code does the}
        MY0=28180;                      {pre-emp, up-scale and trunc}
        MX0=DM(msr);
        AY1=H#FFF8;                     {Same effect as down/up shift}
        MX1=H#80;                       {Used for unbaised rounding}
        MR=MX1*MF (SS);                 {Pre-load MR}
        CNTR=window_length;
{       DO post_process UNTIL CE;}
gsm34:     MR=MR+MX0*MY0 (SS), AY0=DM(I0,M0)   {De-emphasis filtering}
           AR=MR1+AY0;
           AF=PASS AR, MX0=AR;
           AR=AR+AF;                    {Upscale output}
           AR=AR AND AY1;               {Spec does this with shifts}
post_process:     DM(I0,M1)=AR, MR=MX1*MF (SS);
{?} IF NOT CE JUMP gsm34;
        DM(msr)=MX0;

   {At this point, the buffer sr can be output to the speaker}

        DIS AR_SAT;
        RTS;                            {Return from Decoder}

{_________________________Subroutines for Decoder____________________________}

recv_data:  I0=DM(recv_buffer);         {This subroutine copies data}
{       DO recv UNTIL CE;}              {from the input coefficient}
gsm35:     AX0=DM(I0,M1);               {buffer to the appropraite }
recv:            DM(I1,M1)=AX0;         {location in memory while}
{?} IF NOT CE JUMP gsm35;
        DM(recv_buffer)=I0;             {maintaining pointer}
        RTS;

setup_wtr:  I5=^drp+120;                {This subroutine copies the}
        CNTR=40;                        {current sub-window data into}
{       DO copy_drp UNTIL CE;}          {a temporary array.  This temp}
gsm36:     AX0=DM(I5,M5);               {array will be used by the}
copy_drp:        DM(I3,M1)=AX0;         {short term synthesis filter}
{?} IF NOT CE JUMP gsm36;
        RTS;
```

(listing continues on next page)

```
{  This section of code computes the short term synthesis filter as
   described in section 4.3.4 of the recommendation}

st_synthesis:MX1=H#80;              {Used in un-biased rounding}
         M0=-3;                     {M0 is changed for this routine}
{        DO st_synth_compute UNTIL CE;}
gsm37:      I5=^rp+7;               {Point to coefficient array}
            I2=^v+7;                {Point to delay array}
            MY0=DM(I5,M6);          {Moved from PM}
            MR=MX1*MF (SS), MX0=DM(I2,M2);
            AY0=DM(I1,M1);          {AY0 holds sri, sri=wt[k]}
            CNTR=8;
{           DO st_synth_loop UNTIL CE;}
gsm38:            MR=MR+MX0*MY0 (SS);
                  AY1=MX0, AR=AY0-MR1;                {AR=sri}
                  MR=MX1*MF (SS), AY0=AR;             {AY0=sri}
                  MR=MR+AR*MY0 (SS), MX0=DM(I2,M3);
                  AR=MR1+AY1, MY0=DM(I5,M6);          {AR=v[9-i]} st_synth_loop:
DM(I2,M0)=AR, MR=MX1*MF (SS);                         {Save v[9-i]}
{?} IF NOT CE JUMP gsm38;
            DM(I0,M1)=AY0;                      {sr[k]=sri}
            MODIFY(I2,M3);                      {Move pointer to delay line}
st_synth_compute: DM(I2,M0)=AY0;                {V[0]=sri}
{?} IF NOT CE JUMP gsm37;
         M0=0;                                  {Reset M0 to usual value}
         RTS;

{  This section of code computes the long term synthesis filter as
   described in section 4.3.2 of the recommendation}

lt_predictor:AY1=DM(nrp);                       {Check the limits of Ncr}
         AR=DM(Nc);
         AY0=40;
         AF=AR-AY0;
         IF LT AR=PASS AY1;                     {Below min, so use last value}
         AY0=120;
         AF=AR-AY0;
         IF GT AR=PASS AY1;                     {Above max, so use last value}
         DM(nrp)=AR;
         AY0=^drp+120;
         AR=AY0-AR;
         I4=AR;
         I6=AY0;
         AY0=DM(bc);
         AX0=^table_qlb;
         AR=AX0+AY0;
         I5=AR;
         MX1=H#80;
         MR=MX1*MF (SS), MX0=DM(I4,M5);
         MY0=PM(I5,M4);  {brp}
```

```
        I2=^ep;
        CNTR=sub_window_length;
{       DO compute_drp UNTIL CE;}
gsm39:     MR=MR+MX0*MY0 (ss), AY0=DM(I2,M1);        {Compute drpp}
           AR=MR1+AY0, MX0=DM(I4,M5);                {drp[k]=erp[k]+drpp}
compute_drp:     DM(I6,M5)=AR, MR=MX1*MF (SS);       {Store drp[k]}
{?} IF NOT CE JUMP gsm39;
        I4=^drp;                                     {I0 points to drp[-120]}
        I5=^drp+40;                                  {I1 points to drp[-80]}
        CNTR=120;
{       DO update_drp UNTIL CE;}
gsm40:     AX0=DM(I5,M5);
update_drp:      DM(I4,M5)=AX0;                      {drp[-120+k]=drp[-80+k]}
{?} IF NOT CE JUMP gsm40;
        RTS;

{_________________Common Subroutines for Encoder and Decoder_________________}

{  This section of code decodes the coded log area ratios as defined by
   section 4.2.8 of the recommendation}

decode_LARc:I2=^LARc;
        I1=^LARpp;
        I6=^table_mic;
        I4=^table_inva;
        I5=^table_b;
        SE=1;
        CNTR=8;
{       DO compute_larpp UNTIL CE;}
gsm41:     AX0=DM(I2,M1);
           AY0=PM(I6,M5);
           AR=AX0+AY0, SI=PM(I5,M5);
           SR=LSHIFT AR BY 10 (HI);
           AY1=SR1, SR=LSHIFT SI (HI);   {AY1=temp1}
           AR=AY1-SR1, MY0=PM(I4,M5);    {AR=temp1=temp1-temp2}
           MR0=H#8000;MR1=0;             {Unbiased rounding}
           MR=MR+AR*MY0 (ss);            {MR1=temp1=INVA[i]*temp1}
           AY0=MR1;
           AR=MR1+AY0;                   {AR=LARpp[i]}
compute_larpp:   DM(I1,M1)=AR;           {Store LARpp[i]}
{?} IF NOT CE JUMP gsm41;
        RTS;
```

(listing continues on next page)

```
{  This section of code computes the mantissa and exponent parts of the
   xmaxc coefficient as described in section 4.2.15 of the recommendation}

get_xmaxc_pts:AR=DM(xmaxc);
         AY0=AR;
         AX0=15;
         SR=ASHIFT AR BY -3 (HI);
         AY1=1;
         AR=SR1-AY1;
         AF=AY0-AX0;
         IF LE AR=PASS 0;
         SR=LSHIFT AR BY 3 (HI);
         AY1=AR, AR=AY0-SR1;
         IF NE JUMP else_clause;         {Check if mant==0}
         AY1=-4;                         {Yes, so set mant and ex}
         AR=15;
         JUMP around_else;               {Jump over else_clause}

else_clause:AY0=7;
         AF=AR-AY0;
         CNTR=3;
{        DO normalize_mant UNTIL CE;}
gsm42:      IF GT JUMP normalize_mant;
            SR=LSHIFT AR BY 1 (HI);
            AR=AY1-1;                    {Decrement exponent}
            AY1=AR, AF=PASS 1;           {AY1=exp}
            AR=SR1+AF;                   {Increment mantissa}
normalize_mant:   AF=AR-AY0;
{?} IF NOT CE JUMP gsm42;

around_else:AY0=8;
         AR=AR-AY0;
         MX0=AR;                         {Mant must also be in MX0}
         RTS;

{  This section of code computes the reflection coefficients for the
   interpolated LARs as defined in section 4.2.9.2 of the recommendation}

make_rp: MX1=I6;                         {store I6}
         I5=^LARp;
         I6=^rp;
         CNTR=8;
{        DO compute_rp UNTIL CE; }
gsm43:      AX0=DM(I5,M5);
            AR=ABS AX0;
            AX1=AR;
            SR=LSHIFT AR BY 1 (HI);
            AX0=SR1;                     {AX0=temp<<1}
            SR=ASHIFT AR BY -2 (HI);
            AY0=26112;
```

```
            AR=SR1+AY0;                    {AR=temp>>2 + 26112}
            AY0=20070;
            AY1=11059;
            AF=AX1-AY0;
            IF LT AR=AX1+AY1;              {AR=temp+11059}
            AF=AX1-AY1;
            IF LT AR=PASS AX0;
            IF NEG AR=-AR;                 {Compute sign}
compute_rp:       DM(I6,M5)=AR;            {Store rp[i], Moved from PM}
{?} IF NOT CE JUMP gsm43;
         I6=MX1;                           {restore I6}
         RTS;

{  This section of code computes the interpolation of the LARpp() array
   and calls the subroutine to compute the reflection coefficients, and
   then the appropriate short term filter. This block is defined in section
   4.2.9.1 of the recommendation}

st_filter:  SE=-2;                         {Compute the LARps for }
         I2=I4;                            {k_start = 0 to k_end = 12}
         I3=^LARpp;
         I5=^LARp;
         SI=DM(I3,M1);
         CNTR=8;
{        DO k_end_12 UNTIL CE;}
gsm44:      SR=ASHIFT SI (HI), SI=DM(I2,M1);
            AY0=SR1, SR=ASHIFT SI (HI);
            AF=SR1+AY0;
            SR=ASHIFT SI BY -1 (HI);
            AR=SR1+AF, SI=DM(I3,M1);
k_end_12:         DM(I5,M5)=AR;
{?} IF NOT CE JUMP gsm44;

         CALL make_rp;                     {Compute reflection coeffs}
         CNTR=13;                          {13 filter samples}
         CALL (I6);                        {Analysis or Synthesis}
         I5=^LARp;                         {Compute the LARps for}
         I2=I4;                            {k_start = 13 to k_end = 26}
         I3=^LARpp;
         SE=-1;
         SI=DM(I3,M1);
         CNTR=8;
{        DO k_end_26 UNTIL CE;}
gsm45:      SR=ASHIFT SI (HI), SI=DM(I2,M1);
            AY0=SR1, SR=ASHIFT SI (HI);
            AR=SR1+AY0, SI=DM(I3,M1);
k_end_26:         DM(I5,M5)=AR;
{?} IF NOT CE JUMP gsm45;
```

(listing continues on next page)

```
        CALL make_rp;                {Compute reflection coeffs}
        CNTR=14;                     {14 filter samples}
        CALL (I6);                   {Analysis or Synthesis}
        I5=^LARp;                    {Compute the LARps for}
        I2=I4;                       {k_start = 27 to k_end = 39}
        I3=^LARpp;
        SE=-2;
        SI=DM(I2,M1);
        CNTR=8;
{       DO k_end_39 UNTIL CE;}
gsm46:     SR=ASHIFT SI (HI), SI=DM(I3,M1);
           AY0=SR1, SR=ASHIFT SI (HI);
           AF=SR1+AY0;
           SR=ASHIFT SI BY -1 (HI);
           AR=SR1+AF, SI=DM(I2,M1);
k_end_39:        DM(I5,M5)=AR;
{?} IF NOT CE JUMP gsm46;

        CALL make_rp;                      {Compute reflection coeffs}
        CNTR=13;                           {13 filter samples}
        CALL (I6);
        I5=^LARp;                          {Compute the LARps for}
        I3=^LARpp;                         {k_start = 40 to k_end = 159}
        CNTR=8;
{       DO k_end_159 UNTIL CE;}
gsm47:     AX0=DM(I3,M1);
           DM(I5,M5)=AX0;
k_end_159:       DM(I4,M5)=AX0;            {LARpp(j-1)[i] = LARpp(j)[i]}
{?} IF NOT CE JUMP gsm47;

        CALL make_rp;                      {Compute reflection coeffs}
        CNTR=120;                          {120 filter samples}
        CALL (I6);
        RTS;

{  This section of code computes the inverse of the APCM quantization
   and the RPE grid positioning as defined in sections 4.2.16 and 4.2.17
   of the recommendation}

rpe_decoding:I0=^ep;
        AX0=0;                             {First set output ep() array}
        CNTR=sub_window_length;            {to 0s, so it can be filled}
{       DO zero_fill_ep UNTIL CE;}         {in the next section}
zero_fill_ep:    DM(I0,M1)=AX0;
{?} IF NOT CE JUMP zero_fill_ep;

        AX0=DM(mc);
        AY0=^ep;
        AR=AX0+AY0;
```

```
        I1=AR;                              {Point to start in ep() array}
        M0=3;
        AY0=^table_fac;
        AX0=MX0;
        AR=AX0+AY0;
        I5=AR;
        MY0=PM(I5,M4);                      {MY0 holds temp1}
        AX0=6;
        AR=AY1-AX0;
        AX1=AR, AF=AX0-AY1;
        AR=AF-1;
        SE=AR, AR=PASS 1;                   {SE holds temp2}
        SR=LSHIFT AR (HI), SE=AX1;
        AY1=SR1;                            {AY1 holds temp3}
        I0=^xmc;
        AY0=7;
        MX1=H#80;
        MR=MX1*MF (SS), SI=DM(I0,M1);
        CNTR=13;
{       DO inverse_apcm UNTIL CE;}
gsm48:     SR=LSHIFT SI BY 1 (HI);
           AR=SR1-AY0, SI=DM(I0,M1);        {AR=temp=xMc[i]<<1 - 7}
           SR=LSHIFT AR BY 12 (HI);         {SR1=temp=temp<<12}
           MR=MR+SR1*MY0 (SS);              {MR1=temp=temp1*temp}
           AR=MR1+AY1;                      {AR=temp=temp+temp3}
           SR=ASHIFT AR (HI);               {xMp[i]=temp>>temp2}
inverse_apcm:     DM(I1,M0)=SR1, MR=MX1*MF (SS);   {ep[Mc+(3*i)=xMp[i]}
{?} IF NOT CE JUMP gsm48;

   M0=0;                                    {Reset M0 to usual value}
   RTS;
{________________________End of GSM0610 Code_______________________________}
.ENDMOD;
```

Listing 4.2 Codec Routine (GSM0610.DSP)

4 GSM Codec

```
{_____________________________________________________________________________
GSM0632.DSP
            Analog Devices INC. DSP Division
            One Technology Way, Norwood, MA 02062
            DSP Applications Hotline: (617) 461-3672

   This subroutine implements the voice activity detection algorithm of
   GSM specification 06.32 on the ADSP-210x family of DSPs. In line
   comments reference various sections of this recommendation. It
   is assumed that the reader is familiar with that document.

   The code consists of two subroutines. VAD_ROUTINE is called by
   the GSM encoder (06.10) after the autocorrelation is complete.
   UPDATE_PERIODICITY is called by the GSM encoder after the subwindow
   data is calculated.

   This code is optimized to implement the Voice Activity Detection
   in a minimal amount of Progam Memory space. Since the 21xx processors
   can execute all of the GSM speech processing functions in much less
   than 20ms, we have slightly increased execution time (less than .02ms)
   in exchange for a decrease in code size.

   Long words are stored as two successive 16 bit locations,
   MSW first, LSW second.

   This code has been successfully verified with the GSM 06.32 Digital
   Test Sequences, dated March, 1990. The changes made to version 1.00
   during validation are available in a separate document.

Release History:
___Date___   _Ver_ ___________________Comments___________________
24-Oct-89    66    preliminary - waiting for test vectors
10-Jan-90    1.00  Second Release (waiting for VAD test vectors)
01-Nov-90    2.00  Third release. Validated with 06.32 test sequences

Information furnished by Analog Devices is believed to be accurate and
reliable. However, no responsibility is assumed by Analog Devices for its
use; nor for any infringement of patents or other rights of third parties
which may result from its use. Portions of the algorithms implemented in
this code may have been patented; it is up to the user to determine the
legality of their application.

         Assembler Preprocessor Switches:
   -cp switch must always be used when assembling
   -Dalias switch aliases some variables to save RAM space

         Calling Parameters:
   M0=0;  M1=1;  M2=-1;  M3=2;  M4=0;  M5=1;  M6=-1;
   L0=0;  L1=0;  L2=0;   L3=0;  L4=0;  L5=0;  L6=0;

         Return Values: VAD
```

```
        Max Loop Nesting Depth: 2 levels
        Max PC Stack Nesting Depth: 3 levels

        Modes Assumed: AR_SAT enabled, M_MODE disabled
        ADSP-2101 Execution cycles:  2141 maximum

           vad_routine:  2055 cycles maximum
           update_periodicity: 86 cycles maximum
_____________________________________________________________________________}

.MODULE  voice_activity_detection;

{__________________________Conditional Assembly_____________________________}
{    Use (asm21 -cp -Dalias) to alias some variables to save RAM       }
#ifdef alias
   .INCLUDE       <var0632.ram>;
   .EXTERNAL            wt;                     {Working buffer for aliases}
   #define r_a_av1 wt+0
   #define vpar wt+0
   #define sacf wt+9
   #define sav0 wt+9
   #define L_coef wt+18
   #define L_av0 wt+36
   #define L_av1 wt+54
   #define L_work wt+54
#else
   .INCLUDE       <var0632.h>;
#endif
{___________________________________________________________________________}
.ENTRY      vad_routine;
.ENTRY      update_periodicity;

.EXTERNAL   schur_routine;                      { found in GSM0610.DSP }
.EXTERNAL   divide_routine;                     { found in GSM0610.DSP }

.EXTERNAL   L_ACF;
.EXTERNAL   scaleauto;

.GLOBAL  vad, lags;
{  the following are GLOBAL for the reset routine only }
.GLOBAL  rvad, normrvad, L_sacf, L_sav0;
.GLOBAL  pt_sacf, pt_sav0, L_lastdm;
.GLOBAL  oldlagcount, veryoldlagcount, e_thvad, m_thvad, adaptcount;
.GLOBAL  burstcount, hangcount, oldlag;
```

(listing continues on next page)

```
{______3.1__________Adaptive Filtering and Energy Computation______________}

{             Test if L_ACF is equal to zero                        }

vad_routine:I6=^L_ACF;
         AR=DM(scaleauto);
         AR=PASS AR, AY0=DM(I6,M5); {Get ms_ACF}
         IF LT AR=PASS 0;           {IF scaleauto<0 THEN: scalvad=0}

         SR=ASHIFT AR BY 1 (LO);
         AY1=SR0;                   {AY1=scalvad<<1}
         AR=PASS 0, AX0=DM(I6,M6);  {Get ls_ACF}
         DM(m_pvad)=AR;             {Init these anyways}
         DM(m_acf0)=AR;
         AR=-32768;
         DM(e_pvad)=AR;
         DM(e_acf0)=AR;
         AR=AX0 OR AY0, MR0=AY0;    {IF L_ACF[0]=0 THEN: goto 3.2}
         IF EQ JUMP acf_average;

{Outputs: scalvad<<1=AY1, ls_ACF[0]=AX0, I6=^L_ACF[0]}

{           Renormalization of the L_acf[0..8]                  }

         SE=EXP MR0 (HI), SI=AX0;   {Norm L_ACF[0]}
         SE=EXP SI (LO);
         AY0=SE;                    {Fix SE for >>19, take SR1}
         AX0=-3;
         AR=AX0-AY0;
         SE=AR;                     {SE=normacf-3}

         I5=^sacf;
         CNTR=9;
         DO norm_sacf UNTIL CE;
            SI=DM(I6,M5);
            SR=ASHIFT SI (HI), SI=DM(I6,M5);
            SR=SR OR LSHIFT SI (LO);
norm_sacf:        DM(I5,M5)=SR1;

{Outputs: scalvad<<1=AY1, -normacf=AY0}

{        Computation of e_acf and m_acf0              }

         I5=^sacf;
         AX0=32;
         AR=AX0+AY1;                {e_acf0=32+(scalvad<<1)+(-normacf)}
         AR=AR+AY0, SI=DM(I5,M5);   {get sacf[0]}
         DM(e_acf0)=AR;
         SR=ASHIFT SI BY 3 (LO);    {m_acf0=sacf[0]<<3}
         DM(m_acf0)=SR0;
```

```
{Outputs: scalvad<<1=AY1, e_acf0=AR, sacf[0]=SI, I5=^sacf[1]}

{        Computation of e_pvad and m_pvad                   }

         AY0=14;
         AF=AR+AY0, MX1=SI;
         AX0=DM(normrvad);               {normrvad is stored as -normvad}
         I0=^rvad;                       {AF will be e_pvad}
         AF=AX0+AF, MY1=DM(I0,M1);       {get rvad[0] ahead of time}
                                         {get rvad[1], sacf[1]}
         MR=MX1*MY1 (SS), MX0=DM(I0,M1);  {sacf[0]*rvad[0]}
         MY0=DM(I5,M5);
         SR=ASHIFT MR1 BY -1 (HI);       { >> 1}
         SR=SR OR LSHIFT MR0 BY -1 (LO);
         MR0=SR0;
         MR1=SR1;
         CNTR=7;
         DO compute_pvad UNTIL CE;
            MR=MR+MX0*MY0 (SS), MX0=DM(I0,M1);
compute_pvad:     MY0=DM(I5,M5);
         MR=MR+MX0*MY0 (SS);
         AR=PASS MR1, AY0=MR0;
         IF LT JUMP msw_le;              {IF ms_temp>=0}
         AR=AR OR AY0;
         IF NE JUMP gt_zero;             {THEN IF L_temp==0}

msw_le:  MR1=0;                          {THEN: L_temp=1}
         MR0=1;

gt_zero: SE=EXP MR1 (HI);                {SE= -NORM(L_temp)}
         SE=EXP MR0 (LO);

         SR=NORM MR1 (HI);               {L_temp<<normprod, use SR0}
         SR=SR OR NORM MR0 (LO), AR=SE;

         AR=AR+AF;                       {e_pvad-normprod}
         DM(e_pvad)=AR;
         DM(m_pvad)=SR1;

{Outputs: scalvad<<1=AY1}
{______3.2___________ACF Averaging_________________________________________}

acf_average:AX0=-10;
         AR=AX0+AY1;               {Note that SE is neg for >>}
         SE=AR;                    {so SE is -(10-scalvad<<1)}

{Outputs: scalvad<<1=AY1}
```

(listing continues on next page)

```
{           computation of L_av0[0..8] and L_av1[0..8]      }

         L6=72;                     {Circular buffers for L_sav0}
         L3=54;                     {and L_sacf, restore afterwards}
         M2=17;                     {Skip forward 9, 8.5 longs}
         M3=-35;                    {Skip back 17, -17.5 longs}
         I4=^L_ACF;                 {Restore Ms and Ls after use!}
         I0=^L_av0;
         I1=^L_sacf;
         I3=DM(pt_sacf);            {These pointers are updated using}
         I6=DM(pt_sav0);            {automatic circular buffers}
         I5=^L_av1;
         CNTR=9;
         DIS AR_SAT;

            DO acf_sum UNTIL CE;
            SI=DM(I4,M5);                                 {L_temp=L_ACF[i]>>scal}
            SR=ASHIFT SI (HI), SI=DM(I4,M5);
            SR=SR OR LSHIFT SI (LO), AY1=DM(I1,M1);   {Get L_sacf[i]}

            AY0=DM(I1,M2);
            AF=SR0+AY0,   AY0=DM(I1,M1);  {Get L_sacf[i+9]}
            AR=SR1+AY1+C, AX0=DM(I1,M2);
            AF=AX0+AF,    AY1=DM(I1,M1);  {Get L_sacf[i+18]}
            AR=AR+AY0+C,  AX0=DM(I1,M3);  {and skip back 17.5 longs}
            AF=AX0+AF,    DM(I3,M1)=SR1;  {L_sacf[pt_sacf]=L_temp}
            AR=AR+AY1+C,  DM(I3,M1)=SR0;
            AX1=AR, AR=PASS AF;
            DM(I0,M1)=AX1;                {L_av0[i]=sum}
            DM(I0,M1)=AR;

            AX0=DM(I6,M5);                {L_av1[i]=L_sav0[pt_sav0+i]}
            DM(I5,M5)=AX0;
            AX0=DM(I6,M6);
            DM(I5,M5)=AX0;

            DM(I6,M5)=AX1;                {L_sav0[pt_sav0+i]=sum}
acf_sum:    DM(I6,M5)=AR;

         ENA AR_SAT;
         DM(pt_sacf)=I3;                  {Update pointers}
         DM(pt_sav0)=I6;

         L6=0;                            {Restore DAG regs}
         L3=0;
         M2=-1;
         M3=2;
```

```
{______3.3___________Predictor Values Computation____________________________}

{        3.3.1 Schur recursion                              }

         AY1=^L_av1; {in DM}              {Set calling parameters}
         MY1=^sacf;
         M0=^vpar;   {in DM}              {M0 is reset to 0 in subroutine}
         CALL schur_routine;              {Located in 06.10}

{Outputs: none}

{        3.3.2 Step up to obtain aav1[0..8]                 }

         I6=^L_coef;
         I4=^vpar;

         AR=0x2000;                       {MSW 16384<<15}
         DM(I6,M5)=AR;                    {ms_coef[0]=16384<<15}
         AR=PASS 0, SI=DM(I4,M5);         {Get vpar[1]}
         DM(I6,M5)=AR;                    {ls_coef[0]=0}
         SR=ASHIFT SI BY 14 (LO);         {L_coef[1]=vpar<<14}
         DM(I6,M5)=SR1, AR=PASS 1;        {Setup AR as counter}
         DM(I6,M5)=SR0;
         AY0=AR;

{Outputs: AY0=1, AR=m counter=1, I6=^L_coef[2], I4=^vpar[2]}

{        Loop on the LPC analysis order               }

         M3=-2;                           {Restore Ms after use}
         M6=2;
         I5=^L_coef+2;
         CNTR=7;                          {7,6,5,4,3,2,1}
         DO m_loop UNTIL CE;
            I0=^L_coef+2;
            I1=I5;                        {Index for m-i}
            I2=^L_work;
            MODIFY(I5,M6);                {Modify for next time thru}
            SR0=DM(I4,M5);                {Get vpar[m]}
            CNTR=AR;                      {Loop m-1 times}
            DO v_mac UNTIL CE;
                  MR1=DM(I0,M1);          {MR=L_coef[i]}
                  MR0=DM(I0,M1);
                  MY0=DM(I1,M3);          {Get L_coef[m-i]>>16}
                  MR=MR+SR0*MY0 (SS);     {ms_coef[m-i]*vpar[m]}
                  IF MV SAT MR;           {Saturate may not be needed}
                  DM(I2,M1)=MR1;          {L_work=...}
v_mac:            DM(I2,M1)=MR0;
```

(listing continues on next page)

```
            I2=^L_work;                    {L_work starts at [1] not [0]}
            I0=^L_coef+2;
            CNTR=AR;                       {Loop m-1 times}
            DO copy_row UNTIL CE;
                  AX0=DM(I2,M1);
                  DM(I0,M1)=AX0;
                  AX0=DM(I2,M1);
copy_row:               DM(I0,M1)=AX0;

            SR=ASHIFT SR0 BY 14 (LO);      {L_coef[m]=vpar[m]<<14}
            DM(I6,M5)=SR1;
m_loop:     DM(I6,M5)=SR0, AR=AR+AY0;      {Increment m counter}

         M3=2;                             {Restore DAG}
         M6=-1;

{Outputs: none}

{        Keep the aav1[0..8] for next section                    }

         I0=^L_coef;
         I2=^r_a_av1;                      {aav1, rav1 and aav1 are shared}
         SE=-19;
         CNTR=9;
         DO shift_aav1 UNTIL CE;
            SI=DM(I0,M3);
            SR=ASHIFT SI (HI);
shift_aav1:       DM(I2,M1)=SR0;           {aav1[i]=L_coef[i]>>19}


{Outputs: none}

{        Computation of the rav1[0..8]                           }
         I2=^r_a_av1;                      {rav1 here}
         I3=^L_work;
         CNTR=9;
         DO i_loop UNTIL CE;
            I0=^r_a_av1;
            I1=I2;
            MR=0, MX0=DM(I2,M1);           {Modify I2 with dummy read}
            SI=CNTR;
            CNTR=SI;                       {Loop 8-i times}
            DO k_loop UNTIL CE;
                  MX0=DM(I0,M1);
                  MY0=DM(I1,M1);
k_loop:           MR=MR+MX0*MY0 (SS);      {Sum(aav1[k]*aav1[k+i])}
```

```
            DM(I3,M1)=MR1;                {Save L_work[i]}
i_loop:     DM(I3,M1)=MR0;

         I3=^L_work;
         I0=^r_a_av1;

         AR=DM(I3,M1);                    {SE=-NORM(L_work[0])}
         SE=EXP AR (HI), SI=DM(I3,M2);
         SE=EXP SI (LO), AY0=SI;

         AR=AR OR AY0, AX0=SE;
         IF NE AR=PASS AX0;               {IF L_work==0 THEN: AR=SE}
                                          {ELSE: AR=0}
         DM(normrav1)=AR;                       {Save -normrav1 for 3.6}
         SE=AR;                           {Keep -normrav1 for 3.4}
         CNTR=9;
         DO norm_rav1 UNTIL CE;
            SI=DM(I3,M1);
            SR=NORM SI (HI), SI=DM(I3,M1);
            SR=SR OR NORM SI (LO);
norm_rav1:        DM(I0,M1)=SR1;          {rav1[i]=L_work<<normrav1}

{Outputs: -normrav1=SE}

{______3.4___________Spectral Comparison____________________________________}

{        Renormalize L_av0[0..8]                                           }
         I0=^L_av0;
         I1=^sav0;
         CNTR=9;

         SR0=DM(I0,M1);
         AY0=DM(I0,M2);
         AR=SR0 OR AY0, AY1=SE;           {Save -normrav1 in AY1}
         IF NE JUMP else_norm;            {IF sav0==0}

         AR=4095;                         {THEN: sav0[i]=4095}
         DO init_sav0 UNTIL CE;
init_sav0:        DM(I1,M1)=AR;
   JUMP endif_L_av0;
```

(listing continues on next page)

```
else_norm:  SE=EXP SR0 (HI), SI=AY0;      {SE=-shift=NORM(L_av0[0]}
         SE=EXP SI (LO);

         AY0=-3;
         AR=SE;
         AR=AY0-AR;                       {AR=shift-3}
         SE=AR;
         DO norm_av0 UNTIL CE;            {sav0[i]=(L_av0[i]<<shift-3)>>16}
            SI=DM(I0,M1);
            SR=ASHIFT SI (HI), SI=DM(I0,M1);
            SR=SR OR LSHIFT SI (LO);
norm_av0:         DM(I1,M1)=SR1;

{Outputs: -normav1=AY1}

{        Compute partial sum of dm                                }

endif_L_av0:I0=^sav0+1;
         I1=^r_a_av1+1;
         MR=0;                            {L_sump=0}
         CNTR=8;
         DO sump UNTIL CE;
            MX0=DM(I0,M1);
            MY0=DM(I1,M1);
sump:             MR=MR+MX0*MY0 (SS);

{Outputs: -normav1=AY1, L_sump=MR}

{        Compute division of partial sum by sav0[0]               }

         AF=PASS 0;
         AR=ABS MR1, AY0=MR1;             {Set AS flag on L_sump for later}
         IF POS JUMP sump_ge;             {IF L_sump<0}

         DIS AR_SAT;
         AR=AF-MR0;                       {THEN: Negate L_sump}
         ENA AR_SAT;
         MR0=AR, AR=AF-MR1+C-1;
         MR1=AR;

sump_ge: AR=MR0 OR AY0;                   {IF L_temp==0}
         IF NE JUMP sump_ne;

         SE=0;                            {THEN: shift=0}
         MR=0;                            {      L_dm=0}
         JUMP endif_sump;

sump_ne: SI=DM(sav0);
         SR=ASHIFT SI BY 3 (LO);          {AY0=sav0[0]<<3}

         SE=EXP MR1 (HI), AY0=SR0;        {SE=-shift}
         SE=EXP MR0 (LO);
```

```
         SR=NORM MR1 (HI);                  {temp=(L_temp<<shift)>>16}
         SR=SR OR NORM MR0 (LO);

         AF=SR1-AY0, AX0=AY0;               {IF sav0[0]>=temp}
         IF GT JUMP divshift_1;

divshift_0: AF=PASS SR1;                    {THEN: will do temp/sav0[0]}
         AX1=0;                             {      lsw of L_dm=0}
         JUMP endif_sav0;

divshift_1: AX1=32768;                      {ELSE: lsw of L_dm=32768}
                                            {      do (temp-sav0[0])/sav0[0]}
endif_sav0: CALL divide_routine;            {Do divide AY0=AF/AX0}

         AF=PASS 0;
         AX0=0;                             {L_dm+temp, do the <<1 later}
         DIS AR_SAT;
         AR=AX1+AY0;
         SR0=AR, AR=AX0+C;

         IF POS JUMP sump_pos;              {IF L_sump<0, set by abs earlier}

         SR1=AR, AR=AF-SR0;                 {THEN: -L_dm}
         SR0=AR, AR=AF-SR1+C-1;

{Outputs: -normav1=AY1}

{       Renormalization and final computation of L_dm        }

sump_pos:   SR=LSHIFT SR0 BY 15 (LO);       {L_dm<<14+1, do the <<1 here}
         SR=SR OR ASHIFT AR BY 15 (HI);
         AR=SR1, SR=LSHIFT SR0 (LO);        {L_dm=L_dm>>shift}
         SR=SR OR ASHIFT AR (HI);
         MR0=SR0;
         MR1=SR1;

endif_sump: MX0=DM(r_a_av1);                {L_dm+(rav1[0]<<11) with sat}
         MY0=0x0400;                        {For <<11=2^(11-1) and DP add}
         MR=MR+MX0*MY0 (SS), SE=AY1;        {SE=-normav1}
         IF MV SAT MR;                      {Saturate L_dm just in case}

         SR=LSHIFT MR0 (LO);                {L_dm>>normrav1}
         SR=SR OR ASHIFT MR1 (HI);
```

(listing continues on next page)

```
{Outputs: L_dm=SR}

{        Compute difference and save L_dm                         }
         I0=^L_lastdm+1;
         AY0=DM(I0,M2);
         AR=SR0-AY0, AY1=DM(I0,M0);       {L_temp=L_dm-L_lastdm}
         ENA AR_SAT;
         AX0=AR, AR=SR1-AY1+C-1;
         DIS AR_SAT;

         IF NOT AV JUMP exit_sat;         {IF overflow}

         AX0=0x0000;                      {THEN: saturate temp}
         IF LT JUMP exit_sat;             {IF >=0}
         AX0=0xFFFF;                      {THEN: saturate -full scale}

exit_sat:   DM(I0,M1)=SR1;                {L_lastdm=L_dm}
         DM(I0,M0)=SR0;

         IF GE JUMP temp_ge;              {IF L_temp<0}

         AX1=AR, AR=AF-AX0;               {THEN: -L_temp}
         AX0=AR, AR=AF-AX1+C-1;           {Can not overflow}

{Outputs: L_temp=AR AX0}

{        Evaluation of the stat flag                                    }

temp_ge: AY0=3277;                        {L_temp-3277}
         AX1=AR, AR=AX0-AY0;
         ENA AR_SAT;
         AR=AX1-AF+C-1;

         IF GE AR=PASS 0;                 {IF L_temp>=0,THEN: stat=0}
         IF LT AR=PASS 1;                 {              ELSE: stat=1}
         DM(stat)=AR;

{Outputs: none}

{______3.5___________Periodicity detection__________________________________}

         AX0 = DM(oldlagcount);
         AY0 = DM(veryoldlagcount);
         AX1 = 4;
         AR  = 0;                         {AR = ptch = 0}
         AF  = AX0 + AY0;
         AF  = AF - AX1;                  {AF = temp - 4}
         IF GE AR = PASS 1;               {IF GE ptch = 1}
         DM(ptch) = AR;
```

```
{Outputs: none}

{______3.6___________Threshold adaption_____________________________________}

{        Test to find if acf0 < pth                              }

         MR0 = 20;                         {MR0 = E_PLEV}
         MR1 = 25000;                      {MR1 = M_PLEV}
         AX0 = DM(e_acf0);
         AY0 = 19;                         {AY0 = E_PTH}
         AR  = AX0 - AY0;
         AR  = PASS AR;
         IF LT JUMP set_thvad;
         IF GT JUMP test_adapt;
         AX0 = DM(m_acf0);
         AY0 = 18750;                      {AY0 = M_PTH}
         AF  = AX0 - AY0;
         IF GE JUMP test_adapt;
set_thvad:  DM(e_thvad) = MR0;             {comp = 1}
         DM(m_thvad) = MR1;
         JUMP vvad_decision;               {jump to section 3.7}

{        Test to find if adaptation is needed                    }

test_adapt: AX0 = DM(ptch);                {comp = 0}
         AY0 = DM(stat);
         MR  = 0;
         AF  = PASS AX0;
         IF NE JUMP clr_adaptcount;
         AF  = PASS AY0;
         IF NE JUMP inc_adaptcount;
clr_adaptcount: DM(adaptcount) = MR0;      {comp = 1}
         JUMP vvad_decision;               {jump to section 3.7}

{        Increment adaptcount                                    }

inc_adaptcount: AY0 = DM(adaptcount);      {comp = 0}
         AY1 = 8;
         AR  = AY0 + 1;
         DM(adaptcount) = AR;
         AF  = AR - AY1;                   {AF = adaptcount - 8}
         IF LE JUMP vvad_decision;         {jump to section 3.7}
```

(listing continues on next page)

```
{       Compute (thvad - thvad/dec)                                    }

        SE  = -5;
        AY1 = 16384;
        SI  = DM(m_thvad);
        SR  = ASHIFT SI (HI), AY0 = SI;
        AR  = AY0 - SR1;                {AR=m_thvad - (m_thvad>>5) }
        AY0 = DM(e_thvad);
        AF  = AR - AY1, SR1 = AR;       {AF=m_thvad-16384, SR1=m_thvad}
        SE  = 1;
        IF LT SR = ASHIFT SR1 (HI);
        AR  = AY0;
        SI  = SR1;                      {SI = m_thvad}
        IF LT AR = AY0 - 1;             {AR = e_thvad}

{outputs: m_thvad=SR1,SI;e_thvad=AR;}

{       Compute (pvad * fac)                               }

        SE  = -2;                       {shift >> 1 and format adjust}
        MX0 = 3;
        MY0 = DM(m_pvad);
        AY1 = DM(e_pvad);
        MR  = MX0 * MY0 (SS), AY0 = SI; {AY0 = m_thvad}
        SR  = LSHIFT MR0 (LO), MR0 = AR; {MR0 = e_thvad}
        AR  = AY1 + 1;                  {AR = e_temp}
        SR  = SR OR ASHIFT MR1 (HI), AY1 = AR;  {SR=L_temp, AY1=e_temp}
        AF  = PASS SR0;                 {L_temp can overflow 1 bit max}
        IF GE JUMP test_thvad;
        SR  = LSHIFT SR0 BY -1 (LO);    {SR0 = m_temp}
        AR  = AY1 + 1;                  {AR=e_temp}

{outputs:m_thvad=AY0,SI;e_thvad=MR0;m_pvad=MY0;m_temp=SR0;e_temp=AR}

{        Test to find if (thvad < pvad*fac)                            }

test_thvad: AY1 = MR0;                  {AY1 = e_thvad}
        MR1 = AR;                       {MR1=e_temp}
        AF  = AY1 - AR, AX0 = SR0;      {AF=e_thvad-e_temp}
        IF LT JUMP compute_min;
        IF GT JUMP pvad_margin;

        AF  = AY0 - SR0;                {AF=m_thvad-m_temp}
        IF GE JUMP pvad_margin;

{outputs:m_temp=SR0,AX0;e_temp=AR,MR1;m_thvad=AY0,SI;e_thvad=MR0,AY1;m_pvad=MY0}

{       Compute minimum [comp=1]                                }
```

```
compute_min:SR  = ASHIFT SI BY -4 (HI);         {SR1=m_thvad >> 4}
         DIS AR_SAT;
         AR  = SR1 + AY0;                       {AR = L_temp}
         ENA AR_SAT;
         AY0 = AR;
         IF NOT AV JUMP update_m_thvad;
         SR  = LSHIFT AR BY -1 (HI);            {SR1 = L_temp >> 1}
         AR  = AY1 + 1, AY0 = SR1;              {AR=ethvad+1,AY0=mthvad}
         AY1 = AR;                              {AY1 = e_thvad}
update_m_thvad: AF  = MR1 - AY1;                {AF = e_temp - e_thvad}
         IF GT JUMP pvad_margin;
         IF LT JUMP update_e_m;
         AF  = AX0 - AY0;                       {AF = m_temp - m_thvad}
         IF GE JUMP pvad_margin;
update_e_m: AY1 = MR1;                          {comp2=1, AY1 = e_thvad}
         AY0 = AX0;                             {AY0 = m_thvad}

{outputs:e_thvad=AY1; m_thvad=AY0; m_pvad=MY0}

{        Compute (pvad + margin) [comp=0,comp2=0]                       }

pvad_margin:DM(e_thvad) = AY1;
         DM(m_thvad) = AY0;
         AY0 = DM(e_pvad);
         MR1 = 19531;                           {MR1 = M_MARGIN}
         MR0 = 27;                              {MR0 = E_MARGIN}
         AR  = MR0 - AY0, AY1 = MY0;            {AR=E_MARGIN-e_pvad, AY1=m_pvad}
         IF EQ JUMP epvad_eq;
         IF LT JUMP epvad_greater;
         swap_values: AR  = -AR, AX0 = AY1;     {MR1 = m_pvad}
         AY0 = MR0;                             {AY0 = E_MARGIN}
         AY1 = MR1;                             {AY1 = M_MARGIN}
         MR1 = AX0;
epvad_greater: SE  = AR;                        {AR  = -temp}
         SR  = ASHIFT MR1 (HI);                 {SR1 = temp}
         DIS AR_SAT;
         AR  = SR1 + AY1;                       {AR  = L_temp}
         ENA AR_SAT;
         SR1 = AR;                              {SR1 = m_temp}
         SE  = -1;
         IF AV SR = LSHIFT AR (HI);             {m_pvad > 0 always}
         AR  = AY0;
         IF AV AR = AY0 + 1;                    {AR  = e_temp}
         JUMP test_for_greater;
epvad_eq:   DIS AR_SAT;
         AR  = MR1 + AY1;                       {AR = m_pvad + M_MARGIN}
         ENA AR_SAT;
         SR  = LSHIFT AR BY -1 (HI);            {SR1 = m_temp}
         AR  = AY0 + 1;                         {AR  = e_temp}
```

(listing continues on next page)

```
{outputs: m_temp=SR1; e_temp=AR}

{        Test to find if (thvad > (pvad+margin))                        }

test_for_greater:
         AY0 = DM(e_thvad);
         AY1 = DM(m_thvad);
         AF  = AY0 - AR;                  {AF = e_thvad-e_temp}
         IF GT JUMP update_thvad;
         IF LT JUMP update_rvad;
         AF  = AY1 - SR1;                 {AF = m_thvad-m_temp}
         IF LE JUMP update_rvad;
update_thvad:DM(e_thvad) = AR;            {comp = 1}
         DM(m_thvad) = SR1;

{outputs: NONE}

{        Initialize new rvad                                      }

update_rvad:MX0 = DM(normrav1);           {comp = 0}
         DM(normrvad) = MX0;
         I0  = ^rvad;
         I1  = ^r_a_av1;                  {rav1, shared by rav1 and aav1}
         CNTR = 9;
         DO write_rvad UNTIL CE;
            MX0 = DM(I1,M1);
write_rvad:       DM(I0,M1) = MX0;

{outputs: NONE}

{        Set adaptcount                                           }

         MX0 = 9;
         DM(adaptcount) = MX0;

{______3.7___________VAD decision___________________________________________}

vvad_decision:  AY0 = DM(e_pvad);
         AY1 = DM(m_pvad);
         AX0 = DM(e_thvad);
         AX1 = DM(m_thvad);
         AR  = AY0 - AX0;
         IF EQ AR = AY1 - AX1;
         AR  = PASS AR;
         AR  = 0;
         IF GT AR = PASS 1;

{outputs: vvad=AR}
```

```
{______3.8___________VAD hangover decision___________________________________}

         AY1 = DM(hangcount);
         AY0 = DM(burstcount);
         AX0 = AR, AR  = PASS 0;          {AX0 = vvad}
         AF  = PASS AX0;
         IF NE AR = AY0 + 1;              {AR = burstcount}
         MR1 = 5;
         AY0 = 3;
         AF  = AR - AY0;
         IF GE AR = PASS AY0;             {AR  = burstcount}
         DM(burstcount) = AR;
         AF  = PASS AF, AR = AY1;
         IF GE AR = PASS MR1;
         AF  = ABS AR, AY1 = AR;
         IF POS AR = AY1 - 1;
         MR1 = AR, AR = PASS AX0;         {MR1 = hangcount}
         IF POS AR = PASS 1;              {AR = vad}
         DM(hangcount) = MR1;
         DM(vad) = AR;
         RTS;                             {Return to Main Speech transcoder}

{outputs: NONE}

{______3.9___________Periodicity updating___________________________________}

update_periodicity:
         AR  = 0;                         {lagcount = 0}
         AY0 = DM(oldlag);
         I1  = ^lags;
         CNTR = 4;
         DO update_lagcount UNTIL CE;
            AX1 = DM(I1,M1);              {AX1=lags[i],AF=oldlag-lags[i],}
            AF  = AY0 - AX1, AY1 = AR;    {AY1=lagcount}
            IF GT JUMP case_1;
case_2:     AR  = PASS AX1;
            JUMP find_smallag;
case_1:     AR = PASS AY0, AY0 = AX1;     {AY0 = minlag, AR = maxlag}
find_smallag:     CNTR = 3;               {AR = smallag}
            DO compute_smallag UNTIL CE;
```

(listing continues on next page)

```
                  AF  = AR - AY0;
compute_smallag:        IF GE AR = PASS AF;     {AR = smallag}
            AF  = AY0 - AR;                     {AF = temp}
            AF  = AF - AR;                      {AF = temp - smallag}
            IF LT AR = AY0 - AR;
            AY0 = 2;
            AF  = AR - AY0, AR = AY1;
            IF LT AR = AY1 + 1;                 {AR=lagcount}
update_lagcount:AY0 = AX1;                      {AY0=oldlag}
         DM(oldlag) = AY0;
         AX0 = DM(oldlagcount);
         DM(oldlagcount) = AR;
         DM(veryoldlagcount) = AX0;
         RTS;                             {Return to main speech transcoder}
.ENDMOD;
```

Listing 4.3 Voice Activity Detection Routine (GSM0632.DSP)

```
{______________________________________________________________________________
GSM_SID.DSP
            Analog Devices Inc.  DSP Division
            One Technology Way, Norwood, MA, 02062
            DSP Applications: (617) 461-3672

   This code generates comfort noise as specified in GSM recommendation
   6.31, section 3.1. Interpolation of the generated values over
   several frames is not implemented.

   This subroutine is called from the dmr_decode routine when the
   frame to be decoded contains comfort noise parameters (silence
   descriptor frame). The frame of coefficients is over-written
   with the necessary LTP gain and lag values, and the pseudo-randomly
   generated grid position and RPE pulses, for each subwindow. The
   program then returns this properly formatted comfort noise frame
   for normal decoding.

   The pseudo-random number generator is adapted from the one found in
   Analog Devices DSP Applications Handbook 1, section 4.6.

   The pseudo-random number generator is also used by the substitution
   and muting sections of GSM_DTX.DSP.

   ADSP-2101 Execution cycles:  379 maximum

Release History:
___Date___  _Ver_ _______________Comments___________________________
24-Aug-89   57    Incorporated random number generator
10-Jan-90   1.00  Second Release
01-Nov-90   2.00  Third release
______________________________________________________________________________}

.MODULE            Generate_Comfort_Noise;

.ENTRY             comfort_noise_generator, make_random;

.VAR/DM/RAM        seed_lsw, seed_msw;

.GLOBAL            seed_lsw, seed_msw;

{*****  This code generates comfort noise as specified in GSM recommendation
     6.31, section 3.1.  Interpolation of the generated values over several
     frames is not implemented. This code can be further optimized for
     the ADSP-2101. *****}
```

(listing continues on next page)

```
comfort_noise_generator:

         M3  = 8;                   {I1 holds pointer to coeff}
         MODIFY(I1,M3);             {Skip LAR values}
         M3  = 2;                   {Reset M3}

         MX0 = 40;
         MX1 = 120;                 {Constants to write to buffer}
         MY1 = 25;                  {Upper half of a}
         AX0 = 26125;               {Lower half of a}
         SE  = -1;
         SR0 = DM(seed_lsw);
         SR1 = DM(seed_msw);

{For random numbers in the range:   0 to 3      AX1 = 0, MY0 = 2
                                    1 to 6      AX1 = 1, MY0 = 3}

         CNTR = 2;
         DO cn_update UNTIL CE;

            DM(I1,M1) = MX0;        {LTP lag (Ncr) }
            AR  = PASS 0;
            DM(I1,M1) = AR;         {LTP gain (bcr) }

            AX1 = 0;
            MY0 = 2;
            CNTR = 1;
            CALL make_random;       {RPE grid position (Mcr) }

            MODIFY(I1,M1);          {skip block amplitude (Xmaxcr) }

            AX1 = 1;
            MY0 = 3;
            CNTR = 13;
            CALL make_random;       {RPE pulses 1 to 13 (Xmcr) }

            DM(I1,M1) = MX1;        {LTP lag (Ncr) }
            AR  = PASS 0;
            DM(I1,M1) = AR;         {LTP gain (bcr) }

            AX1 = 0;
            MY0 = 2;
            CNTR = 1;
            CALL make_random;       {RPE grid position (Mcr) }

            MODIFY(I1,M1);          {skip block amplitude (Xmaxcr) }

            AX1 = 1;
            MY0 = 3;
            CNTR = 13;
            CALL make_random;       {RPE pulses 1 to 13 (Xmcr) }
```

```
cn_update:        DM(seed_lsw) = SR0;
         DM(seed_msw) = SR1;

         RTS;

make_random:DO gen_random UNTIL CE;
            MR  = SR1 * MY0 (UU);                     {Scale the seed}
            AY0 = MY0;
            AY1 = MR1;                                {Scaled seed in AY1}
            MR  = SR0 * MY1 (UU), MY0 = AX0;          {MR = x(lo) * a(hi)}
            MR  = MR + SR1 * MY0 (UU);                {MR = MR + x(hi)*a(lo)}
            AR  = PASS MR1, MR1 = MR0;
            MR2 = AR, AR  = AX1 + AY1;                {Offset the scaled seed}
            MR0 = H#FFFE;
            MR  = MR + SR0 * MY0 (UU), DM(I1,M1)=AR;  {MR=MR+x(lo)*a(lo)}
            SR  = ASHIFT MR2 BY 15 (HI);
            SR  = SR OR LSHIFT MR1 (HI);
gen_random:       SR  = SR OR LSHIFT MR0 (LO), MY0 = AY0;

         RTS;

.ENDMOD;
```

Listing 4.4 Comfort Noise Insertion Routine (GSM_SID.DSP)

4 GSM Codec

```
{___________________________________________________________________________
GSM_DTX.DSP
            Analog Devices Inc.  DSP Division
            One Technology Way, Norwood, MA  02062
            DSP Applications: (617) 461-3672

   This module contains routines for decoding a codeword that precedes the
   76 coefficients, classifying the frame, performing substitution and
   muting if necessary, and preparing the coefficients for decoding.

   The code is to be executed after the coefficient transfer is complete.
   It assumes that the coefficient buffer was overwritten only with
   GOOD SPEECH or VALID SID parameters. The code executes in the primary
   register set, before the dmr_decode routine is called.

   The 2-bit codeword classifies the frame as follows:
         00 — frame contains speech
         01 — unusable frame
         10 — frame contains valid comfort noise parameters (silence
              descriptor (SID) frame)
         11 — invalid silence descriptor frame - substitute with previous
              valid silence descriptor frame

   ADSP-2101 Computation Time:                 199 cycles maximum.

         state:                         max. cycles
         Good speech                        15
         Valid silence frame                39
         Invalid silence frame              42
         Unusable frame                     199

Release History:
__Date___    _Ver_ _______________Comments_________________________
01-Nov-89    67    Initial implementation
10-Jan-90    1.00  Second Release
01-Nov-90    2.00  Third release
___________________________________________________________________________}

.MODULE            dtx_routine;
.VAR/PM/RAM/CIRC   sil_fram_subwin[17];    { silence frame coeffs (06.11)}
.VAR/PM/RAM        sil_fram_lar[8];        { silence frame coeffs (06.11)}
.VAR/DM/RAM        valid_sid_buffer[9];    { valid coeffs from prior SID}
.VAR/DM/RAM        sub_n_mute;             { flag}
.VAR/DM/RAM        sid_inbuf;              { flag}
.VAR/DM/RAM        taf_count;              { counts frames between valid SID
                                             coeffs during Comfort Noise
                                             Insert}
```

```
.EXTERNAL         make_random;
.EXTERNAL         seed_lsw, seed_msw;

.GLOBAL           sid_inbuf;
.GLOBAL           valid_sid_buffer;
.GLOBAL           sub_n_mute;
.GLOBAL           taf_count;

.ENTRY            decode_codeword;

{these are constants located in program memory ROM}
.INIT       sil_fram_subwin : H#2800, 0, H#100, 0, H#300, H#400, H#300,
                              H#400, H#400, H#300, H#300, H#300, H#300,
                              H#400, H#400, H#300, H#300;
                              {40, 0, 1, 0, 3, 4, 3, 4,
                              4, 3, 3, 3, 3, 4, 4, 3, 3;}
.INIT       sil_fram_lar :    H#2A00, H#2700, H#1500, H#A00, H#900,
                              H#400, H#300, H#200;
                              {42, 39, 21, 10, 9, 4, 3, 2;}

decode_codeword:I0  = ^valid_sid_buffer;
         AY0 = 2;
         MX1 = 1;
         MX0 = -24;
         MY0 = 0;

         AF  = PASS 1, AX0 = DM(I1,M1);   {AX0 = codeword}
         I4  = I1;                        {I4 is working pointer, save I1}
         AR  = AX0 AND AF;
         IF NE JUMP not_good_frame;

good_frame: DM(sub_n_mute) = MY0;
         DM(taf_count) = MX0;

         AR  = AX0 AND AY0;
         IF NE AR = PASS 1;

valid_sid:  DM(sid_inbuf) = AR;
         IF EQ RTS;                       {If good speech, return}
         CNTR = 8;
         M7  = 3;
         DO fill_valid_sid UNTIL CE;
            AR  = DM(I4,M5);
fill_valid_sid:   DM(I0,M1) = AR;         { save LAR values}

         MODIFY (I4,M7);
         M7  = 0;
         AR  = DM(I4,M4);
         DM(I0,M0) = AR;                  { save xmax value}
         RTS;
```

(listing continues on next page)

```
not_good_frame: AR  = AX0 AND AY0;        { At this point, either UNUSABLE or}
         IF NE JUMP invalid_sid;          { INVALID SID frame}

unusable_frame: AX0 = DM(sub_n_mute);
         AX1 = DM(sid_inbuf);
         AF  = PASS AX0;
         IF NE JUMP check_xmax;     {JUMP if NOT first consecutive UNUSABLE}

         AF  = PASS AX1;
         IF EQ JUMP set_subnmut;    {JUMP if not generating comfort noise}

         AY0 = DM(taf_count);
         AF  = PASS AY0;
         IF LE JUMP inc_taf;        {JUMP if waiting for VALID SID frame}

set_subnmut:DM(sub_n_mute) = MX1;
         RTS;

inc_taf: AR  = AY0 + 1;
         DM(taf_count) = AR;
         RTS;

check_xmax: AF  = PASS 0;           { substitution and muting begins}
         M7  = 11;
         MODIFY (I4,M7);            { set pointer to xmax[1]}
         M7  = 17;
         AY0 = 4;
         CNTR = 4;
         DO dec_xmax UNTIL CE;
            AX0 = DM(I4,M4);
            AR  = AX0 - AY0;        { decrement xmax by 4}
            IF GE AF = PASS 1;
            IF LT AR = PASS 0;      { set minimum}
dec_xmax:         DM(I4,M7) = AR;   { write xmax}

         AR  = PASS AF;
         IF NE JUMP not_sil_frame;

writ_sil_frame:DM(sid_inbuf) = AR;  { if all four xmax < 4, insert silence}
         I0  = I1;
         I4  = ^sil_fram_lar;
         CNTR = 8;
         DO writ_sil_lar UNTIL CE;
            AR = PM(I4,M5);
writ_sil_lar:     DM(I0,M1) = AR;
         I4  = ^sil_fram_subwin;
         CNTR = 68;
         L4  = 17;
         DO writ_sil_subwin UNTIL CE;
            AR = PM(I4,M5);
```

```
writ_sil_subwin:  DM(I0,M1) = AR;
         L4  = 0;
         RTS;

not_sil_frame:  AR  = PASS AX1;            { AX1 = sid_inbuf}
         IF NE RTS;                        { if generating comfort noise, grid
                                             position determined elsewhere}
         I4  = I1;
         M1  = 10;                         { set-up}
         MODIFY (I1,M1);
         AX1 = 0;
         MY0 = 2;
         M1  = 17;

         SR0 = DM(seed_lsw);
         SR1 = DM(seed_msw);
         SE  = -1;
         MY1 = 25;
         AX0 = 26125;

         CNTR = 4;
         CALL make_random;
         M1  = 1;
         I1  = I4;
         RTS;

invalid_sid:DM(sub_n_mute) = MY0;          { frame contains INVALID SID parameters}
         DM(taf_count) = MX0;
         DM(sid_inbuf) = MX1;

         CNTR = 8;
         M7  = 3;
         DO writ_valid_sid UNTIL CE;       { replace 8 LARs with previous}
            AR  = DM(I0,M1);               { valid values}
writ_valid_sid:   DM(I4,M5) = AR;
         MODIFY (I4,M7);
         M7  = 17;
         AR  = DM(I0,M0);
         DM(I4,M7) = AR;                   { replace xmax with previous}
         DM(I4,M7) = AR;                   { valid values}
         DM(I4,M7) = AR;
         DM(I4,M4) = AR;
         M7  = 2;
         RTS;

.ENDMOD;
```

Listing 4.5 Discontinuous Transmission Routine (GSM_DTX.DSP)

4 GSM Codec

```
{_____________________________________________________________________________
DMR21xx.DSP

            Analog Devices Inc. DSP Division
            One Technology Way, Norwood, MA 02062
            DSP Applications: (617) 461-3672

   This module is a data acquisition shell for the digital mobile radio
   (GSM) speech processing functions, running on the ADSP-2101 or ADSP-2111
   EZ_LAB. Sound from the microphone input is processed and echoed back to
   the speaker output.

   The interrupt IRQ2 controls the state of the demonstration. There are
   five states, as follows:

   State 0  — input is output directly in a talk-thru mode
            - no encoding, decoding, etc. take place
            - the voice activity flag is disabled

   State 1  — speech is encoded and decoded in a talk-thru mode
            - This mode demonstrates the need for comfort noise
              insertion. The intelligibility of speech in a noisy
              background is reduced.
            - frames are encoded as speech or as comfort noise,
              dependent on the speech flag
            - frames are decoded as speech if the speech flag is
              active, otherwise output is muted
            - the voice activity flag is determined for each frame

   State 2  — speech is encoded and decoded in a talk-thru mode
            - This mode is the normal operation of the GSM system.
            - frames are encoded and decoded as speech or as comfort
              noise, dependent on the speech flag
            - the voice activity flag is determined for each frame

   State 3  — input is encoded and decoded in an example mode
            - each frame is encoded and decoded as a comfort noise
              (silence descriptor) frame
            - the voice activity flag is forced inactive

   State 4  — continuously decodes the last valid silence descriptor frame
              (comfort noise insertion)
            - the voice activity flag is forced inactive

   These five states are cycled through, entering the next state after an
   IRQ2 interrupt. State 0 is the initial state after reset.

   In contrasting states 1 and 2, it is helpful to have a random noise
   source available to mix with the microphone input. This will show the
   adaptation of the voice activity detection threshold, and the loss of
```

```
   intelligibility in state 1 compared to state 2 in a noisy environment.
   The muting in state 1 occurs immediately, unlike the gradual muting
   specified by GSM (which can take up to 320 ms). The code for immediate
   muting is added with the -Ddemo switch.

   The FLAG_OUT signal of the ADSP-2101 or ADSP-2111 EZ_LAB board is
   configured to output the state of the Voice Activity Detector flag in
   states 1 and 2. A high output (LED on) signals that voice activity
   has been detected. This will not work when FLAG_OUT is used to
   control an AD28msp02.

   This implementation allows serial port 0 to accept either 8-bit u-law
   or 16-bit linear data input, based on a C preprocessor switch. The
   u-law hardware companding is used with the codec provided on the
   EZ_LAB board. A 16-bit linear format is used with an AD28msp02
   daughterboard plugged into the codec socket. The default format is
   8-bit u-law.

   This routine takes full advantage of the integration on the ADSP-2101
   and ADSP-2111. It makes use of the IDLE function while waiting for
   the next frame of data. The transfer of the transmit/receive speech
   buffer takes place over serial port 0, using index register I7. If
   using the u-law codec, this is an autobuffered transfer. In order
   for the receive and transmit autobuffering to function synchronously,
   THIS IMPLEMENTATION REQUIRES RFS0 and TFS0 TO BE WIRED TOGETHER
   EXTERNALLY WHEN USING THE u-LAW CODEC. If an AD28msp02 is being used,
   autobuffering is NOT used. THIS IMPLEMENTATION REQUIRES RFS0 and
   TFS0 TO BE SEPARATE WHEN USING THE AD28msp02.

   The Data Address Generator 2 registers I7, L7, M4, and M5 should NEVER,
   NEVER be altered in any routine. They are reserved for input and
   output data buffering, controlled by this shell program.

Release History:
___Date___  _Ver_ _______________Comments_________________________
20-Jun-89   56    Initial release.
04-Jan-90   84    add routine for testing VAD - waiting for vectors
10-Jan-90   1.00  Second release
01-Nov-90   2.00  Third release - added 2111 and 28msp02 capability

         Assembler Preprocessor Switches

   -cp switch           must always be used when assembling
   -Ddemo switch        enables functions necessary for the five-state
                        demonstration
   -Dtestvad            includes code to format coefficients for VAD and
                        SP_FLAG testing
   -Dadsp2111           must be used if running code on the ADSP-2111
                        microcomputer (default is ADSP-2101)
```

(listing continues on next page)

```
   -Dmsp02              changes incoming data format to 16 bit linear for
                        AD28msp02, disables autobuffering (default
                        is u-law codec, autobuffering enabled)

______________________________________________________________________________}

.MODULE/ABS=0     LPC_Codec_Shell;
.VAR/DM/RAM/CIRC  coeff_codeword, coeff_buffer[76];
                                        {Buffer for coeffs, codeword}
.VAR/DM/RAM/CIRC  speech_1[160];
.VAR/DM/RAM/CIRC  speech_2[160];        {Speech windows}

{________________________Conditional Assembly________________________________}
{ use (asm21 -cp -Ddemo) for demonstration }

#ifdef      demo

.VAR/PM/RAM demo_codes[5];              {codes for demonstration only}
.INIT       demo_codes: H#C00000, H#100000, H#200000,
                  H#020100, H#030100;

#endif
{____________________________________________________________________________}

.ENTRY      start_dmr;

.EXTERNAL   dmr_encode, dmr_decode;
.EXTERNAL   reset_codec, decode_codeword;

.EXTERNAL   vad;
.EXTERNAL   sid_inbuf;

.EXTERNAL   sp_flag;
.EXTERNAL   taf_count;                  {temporary - for demonstration}

.GLOBAL     speech_1;
.GLOBAL     speech_2;
.GLOBAL     coeff_codeword;

reset_vector:     JUMP start_dmr; NOP; NOP; NOP;

{________________________Conditional Assembly________________________________}
{ use (asm21 -cp -Ddemo) for demonstration }
#ifdef      demo
irq2:       JUMP next_demo; NOP; NOP; NOP;
#else
irq2:       RTI; NOP; NOP; NOP;
#endif
{____________________________________________________________________________}
```

```
{.......................Conditional Assembly................................}
{ use (asm21 -cp -Dadsp2111) for use with ADSP-2111 }
#ifdef adsp2111
hipw:       NOP; NOP; NOP; NOP;
hipr:       NOP; NOP; NOP; NOP;
#endif
{...........................................................................}

trans0:     RTI; NOP; NOP; NOP;

{.......................Conditional Assembly................................}
{ use (asm21 -cp -Dmsp02) for use with AD28msp02 }
#ifdef msp02
recv0:      JUMP sample; NOP; NOP; NOP;
#else
recv0:      RTI; NOP; NOP; NOP;
#endif
{...........................................................................}

trans1:     NOP; NOP; NOP; NOP;
revc1:      NOP; NOP; NOP; NOP;
timer_int:  NOP; NOP; NOP; NOP;

start_dmr:  ICNTL=B#10100;
            L0=0;   L1=0;   L2=0;   L3=0;
            L4=0;   L5=0;   L6=0;   L7=160;
            M0=0;   M1=1;   M2=-1;  M3=2;
            M4=0;   M5=1;   M6=-1;  M7=0;

            CALL reset_codec;

reg_setup:  AX0 = 0;
            DM(0X3FFE) = AX0;                         { DM wait states }

{.......................Conditional Assembly................................}
{ use (asm21 -cp -Dmsp02) for use with AD28msp02 }
#ifdef msp02

   { initialize 28msp02 - assumes 21xx rfs0, tfs0 separate }
         RESET FLAG_OUT;      { connected to data/~cntl }
         AX0 = 0x2A0F;        { ext sclk, ext rfs, int tfs}
         DM(0x3FF6) = AX0;    { control reg0 }
         AX0 = 0x1000;        { enable serial port0, keep flagout }
         DM(0x3FFF) = AX0;    { system control reg }

         IMASK = 0x10;

         AX0 = 0x20;          { ******* PWDD is inverted in early 28msp02 }
         TX0 = AX0;           { write control word to 28msp02 }
         IDLE;
         AX0 = 0x7C20;
         TX0 = AX0;           { write non-control word to 28msp02 }
```

(listing continues on next page)

```
         IDLE;

         IMASK = 0;
         SET FLAG_OUT;        { connected to data/~cntl }
         AX0 = 0x0000;        { disable serial port0 }
         DM(0x3FFF) = AX0;    { system control reg }

#else

         AX0 = 2;
         DM(0X3FF5) = AX0;    { sclkdiv0 }
         AX0 = 255;
         DM(0X3FF4) = AX0;    { rfsdiv0 }
         AX0 = 0x6927;        { int sclk, int rfs, ext tfs }
         DM(0X3FF6) = AX0;    { control reg0 }
         AX0 = 0X0E77;
         DM(0x3FF3) = AX0;    { autobuffer reg0 }

#endif
{............................................................................}

         I7=^speech_1;        { I7 is speech buffer pointer }

         AX0 = 0X1000;
         DM(0x3FFF) = AX0;    { system control reg }

{________________________Conditional Assembly________________________________}
{ use (asm21 -cp -Ddemo) for demonstration - sets values for state 0}
#ifdef demo
         ENA SEC_REG;
         MR1 = 3; MR0 = 0; MY1 = 0; MY0 = 0; MX1 = 0; SI = 0;
         DIS SEC_REG;
#endif
{____________________________________________________________________________}

{........................Conditional Assembly................................}
{ use (asm21 -cp -Dadsp2111) for use with ADSP-2111 }
#ifdef adsp2111
         IMASK=0x88;
#else
         IMASK=0x28;
#endif
{............................................................................}

{........................Conditional Assembly................................}
{ use (asm21 -cp -Dmsp02) for use with AD28msp02 }
#ifdef msp02
         ENA SEC_REG;
         MX0 = 0;             { reset sample counter }
```

```
        AX1 = 160;           { length of sample buffers speech_1,2 }

code_1_loop:IDLE;                  { wait for next sample }
        AY1 = MX0;
        AR  = AX1 - AY1;           { check if buffer is full }
        IF NE JUMP code_1_loop;

        MX0 = 0;                   { buffer full, reset sample counter }
        DIS SEC_REG;
#else

code_1_loop:IDLE;                  { autobuffering counts samples }
#endif
{..............................................................................}

        I7=^speech_2;              { swap speech output/input buffer }

{________________________Conditional Assembly________________________________}
{ use (asm21 -cp -Ddemo) for demonstration }
#ifdef demo
        ENA SEC_REG;
        AF = PASS MR1;
        IF NE JUMP CODE_2_LOOP;
        M7 = MX1;
        DIS SEC_REG;
#endif
{____________________________________________________________________________}

do_dmr_1:
{.......................Conditional Assembly................................}
{ use (asm21 -cp -Dmsp02) for use with AD28msp02 }
#ifndef msp02
        SE  = 2;                   { left-justify expanded u-law input }
        I0  = ^speech_1;
        CALL scale_routine;
#endif
{..............................................................................}

        I0=^speech_1;
        I1=^coeff_buffer;
        CALL dmr_encode;

{________________________Conditional Assembly________________________________}
{ use (asm21 -cp -Ddemo) for demonstration }
#ifdef demo
```

(listing continues on next page)

```
#ifndef msp02

         CALL vad_out;
#endif
#endif
{________________________________________________________________________________}

         AR  = 2;                   {temporary}
         AX0 = DM(sp_flag);
         AF  = PASS AX0;            {temporary}
         IF NE AR = PASS 0;         {temporary}
         DM(coeff_codeword) = AR;

{________________________Conditional Assembly________________________________}
{ use (asm21 -cp -Dtestvad) to validate VAD and SP_FLAG }
#ifdef testvad
         CALL test_format;
#endif
{________________________________________________________________________________}

         {This is where the coefficient transfer will take place!!}

         I1=^coeff_codeword;
         I2=^speech_1;

{________________________Conditional Assembly________________________________}
{ use (asm21 -cp -Ddemo) for demonstration }
#ifdef demo
         CALL set_codeword;         {routine sets coeff_codeword for demo}
#endif
{________________________________________________________________________________}

         CALL decode_codeword;
         AX0 = DM(sid_inbuf);

{________________________Conditional Assembly________________________________}
{ use (asm21 -cp -Dtestvad) to validate VAD and SP_FLAG }
#ifdef testvad
         CALL test_unformat;
#endif
{________________________________________________________________________________}

         CALL dmr_decode;

{........................Conditional Assembly................................}
{ use (asm21 -cp -Dmsp02) for use with AD28msp02 }
#ifndef msp02
         SE  = -2;                  { right shift to 14 bits for u-law }
```

```
        I0  = ^speech_1;           { compression }
        CALL scale_routine;
#endif
{.............................................................................}

{........................Conditional Assembly................................}
{ use (asm21 -cp -Dmsp02) for use with AD28msp02 }
#ifdef msp02
        ENA SEC_REG;

code_2_loop:IDLE;                        { wait for next sample }
        AY1 = MX0;
        AR  = AX1 - AY1;                 { check if buffer is full }
        IF NE JUMP code_2_loop;

        MX0 = 0;                         { buffer full, reset sample counter }
        DIS SEC_REG;
#else

code_2_loop:IDLE;                        { autobuffering counts samples }
#endif
{.............................................................................}

   I7=^speech_1;                         { swap speech output/input buffer }
{________________________Conditional Assembly________________________________}
{ use (asm21 -cp -Ddemo) for demonstration }
#ifdef demo
        ENA SEC_REG;
        AF  = PASS MR1;
        IF NE JUMP CODE_1_LOOP;
        M7  = MX1;
        DIS SEC_REG;
#endif
{_____________________________________________________________________________}

do_dmr_2:

{........................Conditional Assembly................................}
{ use (asm21 -cp -Dmsp02) for use with AD28msp02 }
#ifndef msp02
        SE  = 2;                         { left-justify expanded u-law input }
        I0  = ^speech_2;
        CALL scale_routine;
#endif
{.............................................................................}

        I0=^speech_2;
```

(listing continues on next page)

```
        I1=^coeff_buffer;
        CALL dmr_encode;

{________________________Conditional Assembly________________________________}
{ use (asm21 -cp -Ddemo) for demonstration }
#ifdef demo
#ifndef msp02
        CALL vad_out;
#endif
#endif
{____________________________________________________________________________}

        AR  = 2;                          {temporary}
        AX0 = DM(sp_flag);
        AF  = PASS AX0;                   {temporary}

        IF NE AR = PASS 0;                {temporary}
        DM(coeff_codeword) = AR;

{________________________Conditional Assembly________________________________}
{ use (asm21 -cp -Dtestvad) to validate VAD and SP_FLAG }
#ifdef testvad
        CALL test_format;
#endif

{____________________________________________________________________________}

        {This is where the coefficient transfer will take place!!}
        I1=^coeff_codeword;
        I2=^speech_2;

{________________________Conditional Assembly________________________________}
{ use (asm21 -cp -Ddemo) for demonstration }
#ifdef demo
        CALL set_codeword;                {routine sets coeff_codeword for demo}
#endif
{____________________________________________________________________________}

        CALL decode_codeword;
        AX0 = DM(sid_inbuf);

{________________________Conditional Assembly________________________________}
{ use (asm21 -cp -Dtestvad) to validate VAD and SP_FLAG }
#ifdef testvad
        CALL test_unformat;
#endif
{____________________________________________________________________________}

   CALL dmr_decode;
```

```
{.......................Conditional Assembly................................}
{ use (asm21 -cp -Dmsp02) for use with AD28msp02 }
#ifndef msp02
         SE = -2;                         { right shift to 14 bits for u-law }
         I0  = ^speech_2;                 { compression }
         CALL scale_routine;
#endif
{...........................................................................}

{.......................Conditional Assembly................................}
{ use (asm21 -cp -Dmsp02) for use with AD28msp02 }
#ifdef msp02
         ENA SEC_REG;                           { sample counting done in sec regs }
#endif
{...........................................................................}

         JUMP code_1_loop;

{________________________Conditional Assembly________________________________}
{ use (asm21 -cp -Ddemo) for demonstration }
#ifdef demo
next_demo:  ENA SEC_REG;
            SE  = 2;
            AY0 = ^demo_codes;
            AR  = SI, AF = PASS 1;
            AY1 = 4;
            AR  = AR + AF;                      {increment current state}
            af  = ar - ay1;
            if gt ar = pass 0;
            SI  = AR, AR  = AR + AY0;           {offset pointer, save state}
            AX0 = I5;
            I5  = AR;
            SR0 = PM(I5,M4);                    {get demo state codeword}
            I5  = AX0;
            ay1 = 7;
            AR  = SR0 AND AY1;                  {extract force_vad_high, _low}
            MX1 = AR, SR = LSHIFT SR0 (LO);     {        talk_thru_flag}
            MR1 = SR1, SR = LSHIFT SR0 (LO);    {        mask_sp}
            MR0 = SR1, SR = LSHIFT SR0 (LO);    {        mask_taf}
            MY1 = SR1, SR = LSHIFT SR0 (LO);    {        force_codeword_high}
            MY0 = SR1;
            RTI;

set_codeword:     ENA SEC_REG;
                  IMASK = 0;
                  AY1 = 3;
                  AF  = PASS 0;
                  AY0 = DM(sp_flag);
```

(listing continues on next page)

```
                  AR  = PASS AY0;
                  IF EQ AF = PASS AY1;
                  AR  = MR0 AND AF;            {AR = masked sp_flag}
                  AY1 = 2;
                  AF  = PASS 1, AX0 = AR;
                  AY0 = DM(taf_count);
                  AR  = PASS AY0;
                  IF GT AF = PASS AY1;
                  AR  = MY1;
         AF  = AR AND AF;                {AF = masked taf_count}
         AF  = AX0 OR AF, AR = MY0;
         AR  = AR OR AF;                 {AR = coeff_codeword}
         DM(coeff_codeword) = AR;
         AY0 = 1;
         AR  = AR - AY0;                 { check if unusable frame }
         IF NE JUMP set_cw_done;
         I4  = I1;                       { unusable frame - force }
         M7  = 12;                       { immediate muting for }
         MODIFY(I4,M7);                  { demonstration by setting }
         M7  = 17;                       { the four xmax values < 4 }
         CNTR = 4;                       { (in this case, = 0) }
         DO set_xmax_demo UNTIL CE;

set_xmax_demo:    DM(I4,M7) = AR;
                  M7  = 2;

{.......................Conditional Assembly................................}
{ use (asm21 -cp -Dadsp2111) for use with ADSP-2111 }
#ifdef adsp2111

set_cw_done: IMASK=0x88;

#else

set_cw_done: IMASK=0x28;

#endif {...............................................}

         DIS SEC_REG;
         RTS;
#endif
{____________________________________________________________________________}

{________________________Conditional Assembly________________________________}
{ use (asm21 -cp -Dtestvad) to validate VAD and SP_FLAG }
#ifdef testvad

test_format:I1  = ^coeff_buffer;
            AX0 = DM(vad);
```

```
            AX1 = DM(sp_flag);
            CNTR = 2;
            DO add_bits UNTIL CE;
                  AR  = H#8000;
                  AF  = PASS AX0, AY0 = DM(I1,M0); IF EQ AR = PASS 0;
                  AR  = AR OR AY0, AX0 = AX1;
                  add_bits: DM(I1,M1) = AR;
            RTS;

test_unformat: AX1 = H#7FFF;
               AY0 = DM(I1,M0);
               AR  = AX1 AND AY0;
               DM(I1,M1) = AR;
               AY0 = DM(I1,M0);
               AR  = AX1 AND AY0;
               DM(I1,M2) = AR;
               RTS;
#endif
{_____________________________________________________________________________}

{________________________Conditional Assembly________________________________}
{ use (asm21 -cp -Ddemo) for demonstration }
#ifdef demo
#ifndef msp02

{this is temporary for outputting the voice activity flag for the demonstration}

vad_out: AX0 = DM(vad);
         AF  = PASS AX0;

         IF NE SET FLAG_OUT;

         IF EQ RESET FLAG_OUT;
         RTS;
#endif
#endif
{_____________________________________________________________________________}
```

(listing continues on next page)

```
{.......................Conditional Assembly................................}
{ use (asm21 -cp -Dmsp02) for use with AD28msp02 }
#ifndef msp02
scale_routine: SI  = DM(I0,M1);
         CNTR = 160;
         DO shift_it UNTIL CE;
            SR  = ASHIFT SI (HI), SI = DM(I0,M2);
shift_it:   DM(I0,M3) = SR1;
         RTS;
#endif
{.......................................................................}

{.......................Conditional Assembly................ .......}
{ use (asm21 -cp -Dmsp02) for use with AD28msp02 }
#ifdef msp02
sample:  ENA SEC_REG;
         AR  = DM(I7,M4);           { read buffer, do not move pointer }
         TX0 = AR;                  { write transmit data }
         AR  = RX0;                 { read received data }
         DM(I7,M5) = AR;            { write to buffer, increment pointer }
         AY0 = MX0;
         AR  = AY0 + 1;             { increment sample counter }
         MX0 = AR;
         RTI;
#endif {..................................................................}

.ENDMOD;
```

Listing 4.6 Data Acquisition Shell Routine (DMR21xx.DSP)